Windowsprogrammering

med Windows SDK/C++

og .NET/C#

Lars Edgar Berg

Teknisk Informatikk AS

Innhold

Forord 4

Microsoft Windows - endel grunnleggende begrep 5
- Operativsystem 5
- Prosess 5
- CUI og GUI 5

Skjermvinduer 6
- Ulike typer vinduer 6
- Vinduene er systemobjekter – identifiseres ved 'hendler' 7

Systemfunksjoner – Win32 API 8
- Noen spesielle datatyper fra Win32 API 9
- Hva er Windows SDK ? 10

Meldingssystemet 11
- MSG 13

Arkitekturen til et grafisk Windowsprogram 15
- Noen viktige meldinger 15

"Hallo Windows" - vårt første GUI-program 16
- Oppretting av et prosjekt i Visual Studio 16
- Kildekode inn i prosjektet 20
- HalloWin – kildekode 21
- En gjennomgang av kildekoden i HalloWin.cpp 23
  - Hovedrutinen - WinMain 23
  - Vindusfunksjonen – WndProc 25

Under overflaten - SPY++ 27
- Øvingsoppgave 1 29

Utskriftsoperasjoner – med tekst og grafikk 31

GDI - Graphics Device Interface 31
- Device Context (utskriftskontekst) 32
- Informasjoner om utstyrsegenskaper 33
- GDI-objekter og Device Context 35
- Utskrift knyttet til WM_PAINT 36
- De viktigste typene av GDI-objekter 37
  - Brush-objekter 37
  - Pen-objekter 38
  - Font-objekter 38
- GDI-Demo – et litt større eksempel 38
- Ferdiglagde GDI-objekter 40

Koordinatsystemer og "mapping mode" 43
- Øvingsoppgave 2 44
  - Kode-eksempel – Sinus.cpp 45

Håndtering av bruker-innputt 47

Mus-hendelser 47
- Andre meldinger trigget av musklikk 48
- Øvingsoppgave 3 48
- Innhold i Oving3.cpp – kode for start av øvingen 49

Timer-mekanismen i Windows 51
Øvingsoppgave 4 52
Tastetrykk 53
Hvilket vindu får tastaturmeldingene? 54
Repeat Count 54
OEM Scan Code 54
Context Code, Previous Key State og Transition State 54
Virtuelle tastaturkoder 54
Tastene SHIFT, ALT og CTRL 55
Andre meldinger fra tastatur-innputt 55
Oppsummering 56

Visuelle elementer i grafiske brukergrensesnitt 58
Ulike vindustyper – relasjoner mellom vinduer 59
Overlappende (Overlapped - opprettet med WS_OVERLAPPED) 59
Eid (Owned window) 59
Popup-vindu (opprettet med WS_POPUP) 59
Child-vindu (opprettet med WS_CHILD) 59
WM_COMMAND 60
Noen standard kontrolltyper 61
Trykknapper 62
Tekstbokser 63
Ressurser og ressurseditorer 64
Eksempel med meny- og dialogressurs 65
Ny menykommando 66
Ny dialog 68
En enkel reaksjonstester 70

Hva er .NET Framework 73
Common Language Runtime (CLR) 74
Virtuell maskin 74
”Loader” og ”runtime system” (kjøresystem) 76
Kompilering Just-In-Time 77
Automatisk "Garbage Collection" 78
Common Type System - CTS 78
Klassifikasjon av datatyper 78
Verdityper/Value types 79
Referansetyper/Reference types 79
Klassebibliotekene i .NET Framework 82
Navnerommet System 82

Forord

Dette kompendiet er utviklet over tid som en støtte til undervisningen i Windowsprogrammering ved Høgskolen i Bergen. Det finnes mange gode lærebøker i Windowsprogrammering, og vi anbefaler studentene å skaffe seg en slik hvis de skal gå dypt og grundig inn i dette området. Men det viser seg ofte at bøkene som er tilgjengelige har visse ulemper i forhold til behovene i våre kurs – det kan være:

- for omfattende med hensyn til detaljer
- for avansert i fremstillingen
- for spesialiserte, dekker ikke alle emnene vi vil inn på

Kompendiet har to hoveddeler. I den første delen tar vi for oss "klassisk" Windowsprogrammering med C++ og ved hjelp av systemfunksjonene i Win32-bibliotekene. Denne delen tar for det første sikte på å gi en grundig beskrivelse av de grunnleggende mekanismene bak det interne kommunikasjonssystemet i Windows – Windows Messages. Videre gjennomgås de viktigste begrepene, operativsystemressursene og datastrukturene som man bør kjenne til for å kunne lage programmer med grafiske brukergrensesnitt.

Den andre delen inneholder en kortfattet og kompakt gjennomgang av den utvidelsen av Windows operativsystem som betegnes .NET Framework samt en innføring i det nye programmeringsspråket C.

Det forutsettes at studentene på forhånd har kjennskap til grunnleggende C++ og til begrepene for objektorientert progamutvikling.

Microsoft Windows - endel grunnleggende begrep

Her inntroduseres en del begrep som er grunnleggende for Microsoft Windows og for svært mange andre moderne operativsystem.

Operativsystem

Et operativsystem er et sett av overordnet programvare i en datamaskin. Hovedoppgavene for operativsystemet er administrasjon av aktiviteter og systemressurser samt å formidle I/O-operasjoner mellom aktivitetenes høynivåfunksjoner og det fysiske utstyret maskinen er utrustet med.

Prosess

Litt kort og upresist kan det sies at en prosess er "et program under kjøring". Mer presist blir det å si at en prosess er den grunnleggende administrative enheten for aktiviteter og ressurser i datamaskinen.

For at operativsystemet skal kunne opprette og starte en prosess må det finnes en eksekverbar programkode for denne – et program. Programfilen inneholder da den maskinkoden som skal utføres og eventuelle "programressurser" – spesielle data som skal benyttes av prosessen.

CUI og GUI

Windows støtter to typer prosesser: CUI-prosesser og GUI-prosesser. CUI står for 'Console User Interface' og GUI er forkortelsen for 'Graphical User Interface'. CUI-prosessene er 'karakterbaserte' – de kjøres i et konsoll-vindu ('DOS-vindu') der all bruker-innputt må komme via tastaturet. I GUI-prosessene får vi et programvindu av den mer moderne typen med innhold av både tekst og grafiske komponenter som knapper, menyer, bilder, m.m.

Det er imidlertid mer enn det ytre brukergrensesnittet som skiller disse to prosesstypene fra hverandre. De har også noe

forskjellige indre arkitekturer, som vi skal komme mer inn på lenger ut i kapitlet.

Skjermvinduer

Helt siden datamaskinene forlot hullkort og perforerte papirstrimler som innputt-mekanismer har de hatt operatør-utstyr bestående av en billedskjerm og et tastatur (de to enhetene tilsammen betegnes ofte som et konsoll). Fra midten av 1980-årene begynte man også å få en peker-mekanisme (vanligvis en "mus") i tillegg til tastaturet for å kunne styre programmene gjennom "pek-og-klikk". Maskinene fikk det som betegnes "et grafisk brukergrensesnitt".

For de enkleste maskinene og programmene kunne operatøren bare arbeide med ett program om gangen. Da ble naturligvis hele billedskjermen benyttet av dette ene programmet. Men etterhvert som maskinene fikk kapasitet til å kjøre flere programmer parallelt, ble det også aktuelt å la hvert program benytte hver sine rektangulære deler av billedskjermen. Hvert program hadde da sitt separate "skjermvindu". Med dagens systemer er dette så vanlig at vanskelig kan tenke oss noe annet.

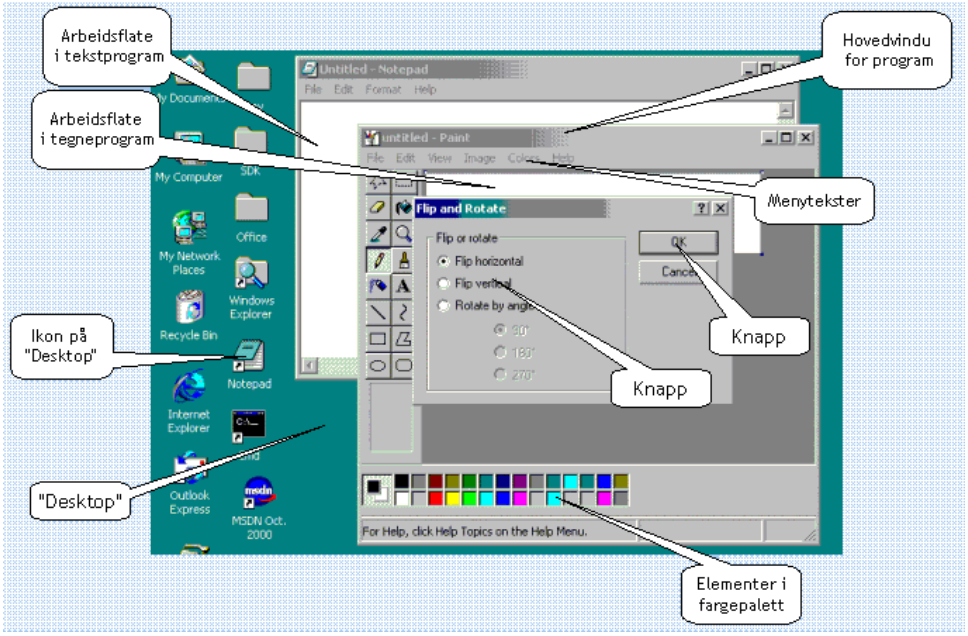
Et skjermvindu er altså en avgrenset rektangulær del av skjermens billedflate. De aller fleste "brukerprogram" (prosesser som er startet av operatøren) har ett eller flere vindu og vinduene spiller en svært sentral rolle i betjeningen av dagens personlige datamaskiner. Ethvert vindu er knyttet til eller "eiet" av en prosess. På dagens systemer kan vinduene overlappe hverandre helt eller delvis. Ett av vinduene og bare ett, vil være det såkalte "aktive" vinduet.

Ulike typer vinduer

Med unntak for visse spesielle program har alle program minst ett vindu – programmets hovedvindu. Dette åpnes automatisk når programmet startes. Under Microsoft Windows kan det opprettes to hovedtyper av programmer – programmer med grafiske hovedvindu og programmer med "gammeldagse" tegnbasert/karakterbasert hovedvindu. Som omtalt tidligere

betegner vi disse programmertypene som GUI- programmer (GUI. Graphical User Interface) og CUI- programmer (CUI: Console User Interface). I resten av dette kurset skal vi bare arbeide med GUI- programmer og grafiske vinduer dersom ikke noe annet sies eksplisitt.

Et grafisk vindu er imidlertid ikke bare et rektangulært område på skjermen. Det finnes en rekke spesielle typer grafiske vindu – fra de ”vanligste” til grafiske symboler av ulike slag. I figuren nedenfor er det vist eksempler på ulike typer grafiske vinduer.



Figuren viser en illustrasjon av endel vindustyper.

Vinduene er systemobjekter – identifiseres ved 'hendler'

På linje med en rekke andre deler av operativsystemet betegnes datastrukturene bak vinduene som "objekter" (systemobjekter). Da menes det ikke "C++-objekter", for operativsystemet er ikke skrevet i C++. Ordet brukes her i en videre forstand om

strukturer som har visse, men begrensede, objektorienterte egenskaper og bruksmåter.

Et systemobjekt blir identifisert i Windows ved en heltallseverdi som tildeles objektet idet det blir opprettet. Denne heltallsverdien betegnes en **objekthendel**. En "hendel" ("handle" på engelsk) er i Windows-sammenheng en variabel av typen HANDLE. Datatypen HANDLE er definert som følger i en av headerfilene for programbiblioteket :

```
typedef HANDLE void*
```

En hendel er altså en generell pekervariabel, men de skal aldri brukes som pekere – de skal bare brukes som en "anonym" 32-bit heltallsverdi. En hendelverdi er bare et slags identitetsnummer for et objekt. Det bare internt i operativsystemet at selve verdien brukes i noen mer spesiell betydning.

Systemfunksjoner – Win32 API

Etter begynnerkurset i C++ kjenner vi alle til bruken av bibliotek-funksjoner og bibliotek-klasser. Dette er "ferdiglaget" kode som følger med programmeringspakken. Funksjonen `strlen(..)` er for eksempel en C++ bibliotekfunksjon som returnerer antall tegn i en char-variabel. Definisjonen av denne funksjonen ligger i headerfilen "string.h" og ferdig kompilert maskinkode for den lenkes inn i vårt program når vi bygger det.

Visual Studio er ikke bare et programmeringsverktøy for standard C++. Det er også et utviklingsverktøy for Windows-programmer, noe som går utover standard C++. Derfor kommer Visual C++ med mange flere bibliotekfunksjoner og –datatyper enn det som er spesifisert kun for C++-språket. Vi kan betegne disse tilleggsmodulene som systembibliotek for Windows-programmering. En annen og mer offisiell betegnelse er "Win32 Application Programming Interface".

Vi skal komme inn på flere detaljer vedrørende Win32 API når vi seinere skal ta fatt på konkrete programmeringsøvinger. La oss bare kort antyde at på tilsvarende måte som vi bruker standard bibliotekfunksjoner, tar vi ibruk Windows system-

funksjoner ved å inkludere en spesiell headerfil og ved å lenke inn den aktuelle maskinkoden for systemfunksjonen.

Noen spesielle datatyper fra Win32 API

Såsnart vi kommer bort i eksempler på Windowsprogrammer, vil vi støte på en rekke tilsynelatende nye datatyper i forhold til de vi kjenner fra standard C++. Noen eksempler:

```
BYTE
UINT
LONG
WORD
DWORD
. . .

HANDLE
HWND
HDC
WPARAM
LPARAM
. . .

POINT
RECT
. . .
```

Noen av disse er "typedef'er" av helt vanlige C++-typer. UINT og WORD er for eksempel definert slik:

```
typedef unsigned int        UINT;
typedef unsigned short      WORD;
```

Slike "omdøpinger" kan synes unødvendig for oss, men Microsoft og andre som vil bruke samme kildekode på data-maskiner med ulike prosessortyper (og dermed for eksempel muligheter for ulike lengder på en C++ "int") trenger dem for lettere å kunne flytte kildekode.

Andre av disse spesialtypene er definert for spesielle Windows-formål. Vi har allerede nevnt datatypen HANDLE som brukes for å identifisere en rekke systemobjekter. Både HWND og HDC er spesielle hendel-typer for henholdsvis vinduer og objekter som kalles "Device Context".

WPARAM og LPARAM er datatyper spesielt brukt for visse parametre i Windows-meldinger. WPARAM var tidliger en WORD-type (16-bits heltall), men måtte oppgraderes til

DWORD (32-bits heltall) med overgangen fra Windows 3.1 (16-bits) til Windows 95 (32-bits). LPARAM har hele tiden vært et 32-bits heltall.

De siste eksemplene – POINT og RECT – er to ordinære C++-struct'er som man ofte har bruk for i grafiske sammenhenger.

```
typedef struct tagPOINT
{
    LONG  x;
    LONG  y;
} POINT;

typedef struct tagRECT
{
    LONG    left;
    LONG    top;
    LONG    right;
    LONG    bottom;
} RECT;
```

Det finnes videre en rekke tilsvarende spesialtyper – og de fleste av dem har navn som består av bare store bokstaver.

Hva er Windows SDK ?

SDK står for Software Development Kit. SDK består av noen hundre bibliotekfunksjoner som utgjør det grunnleggende "fundamentet" for all Windowsprogrammering.

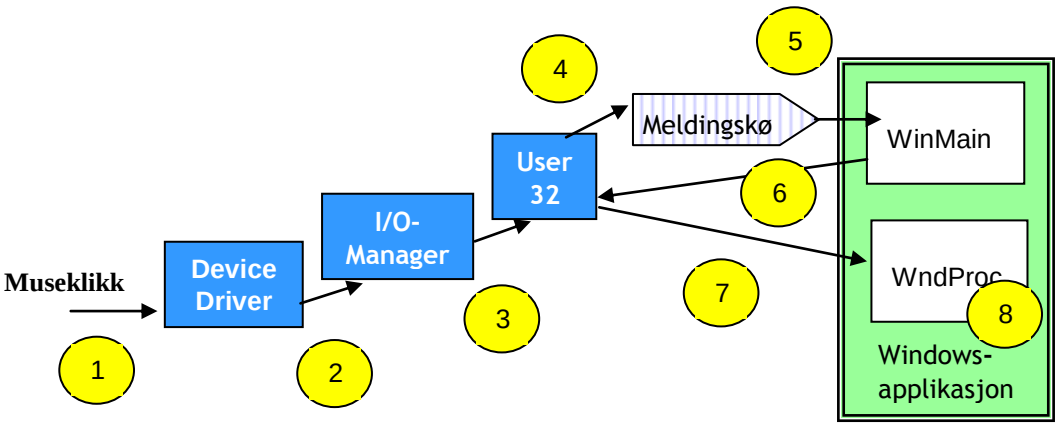
SDK-programmering er den mest fleksible måten å programmere i Windows, den er "nærmest jernet" og gir rask og effektiv kode. Nærheten til operativsystemet med grunnmekanismene gjør det også gunstig å starte med SDK for de som skal forstå Windows svært grundig. Det er riktignok en brattere læringskurve og mer krevende å komme igang med SDK enn med andre verktøy som Visual Basic, Delphi og Visual C++ eller C#. Men det viser seg raskt at et grunnlag i SDK-programmering er svært nyttig uansett hva slags programmeringsverktøy en vil gjøre bruk av seinere.

Meldingssystemet

Microsoft produserer idag flere varianter av operativsystem med betegnelsen Windows "ett eller annet". De kan være nokså ulike med hensyn til intern konstruksjon, men de har alle visse grunnleggende felles egenskaper. De har alle et felles ytre utseende med grafisk brukergrensesnitt i form av "vinduer" og betjening via "pek-og-klikk". Men de har også det til felles at de baserer svært mye av sin interne kommunikasjon på såkalte "Windows-meldinger"/"Windows messages".

Siden et operativsystem utfører tjenester for programmer, og siden kommunikasjonen mellom operativsystem og program er basert på "Windows-meldinger", så er kjennskap til denne grunnleggende mekanismen svært viktig for en som skal lage programvare for Windows-systemer.

Som et eksempel på hvilken funksjon Windowsmeldingene har, skal vi se på hvordan et museklikk fra brukeren formidles gjennom operativsystemet og ut til et Windowsprogram som for eksempel skal tegne en strek ut fra "klikkstedet" på skjermen.



Figuren illustrerer bruken av Windows-meldinger.

1. En brukerhendelse registreres av en device driver.
2. Fra device driver inn i I/O-system.
3. Fra I/O-system til User32 - en modul i operativsystemet.

4. User32 finner ut hvilket program som eier det aktuelle vinduet det er klikket på eller tastet inn til - legger en Windows-melding i programmets meldingskø.

5. GetMessage henter inn meldingen.

6. DispatchMessage ”sender” meldingen tilbake til operativ-systemet for å få kalt den riktige vindusprosedyren. (En prosess kan ha flere vinduer.)

7. Operativsystemet kaller den aktuelle vindusprosedyren med meldingsdata som parametre.

8. Vindusprosedyren utfører den planlagte responsen på brukerens operasjon - eller kaller default-prosedyren.

Når brukeren klikker på musknappen (1) går det en elektrisk impuls inn til datamaskinens "mus-grensesnitt". Dette utstyret overvåkes av en programmodul som vi kan kalle "musdriveren". Betegnelsen "driver" brukes om de modulene som overvåker og styrer datamaskinens "periferi-utstyr", utstyr som kommuniserer med omverdenen på en eller annen måte.

Musdriveren har interne variabler som inneholder mus-pekerens posisjon på skjermen. Den henter det aktuelle tidspunktet fra systemklokken samt tilstanden for visse kontrolltaster (oppe eller nede) og bygger en datapakke med disse informasjonene samlet. Dessuten får denne datapakken en "merkelapp" – for eksempel WM_LBUTTONDOWN dersom det var venstre museknapp som ble trykket. Da har driveren laget en Windowsmelding, og denne blir sendt i første omgang til "overordnede" moduler i operativsystemet – I/O-Manager (2) og User32 (3).

Der suppleres meldingspakken med en informasjon til, nemlig en "vindushendel". Operativsystemet holder nemlig rede på hvilket vindu som ligger under den skjermposisjonen det ble klikket på, og denne informasjonen er viktig både for at operativsystemet skal kunne sende meldingen videre til "rette vedkommende" program, og som en informasjon til det programmet som skal motta meldingen.

Meldingen om museklikket inneholder nå følgende data:

- meldingsnavn (egentlig et nummer – WM_LBUTTONDOWN representerer for eksempel 0x0201 eller 513)
- tidspunktet for klikket
- skjermkoordinatene for klikket
- identifikasjon av vinduet under klikkposisjonen
- dessuten informasjon om kontrolltastenes status, f.eks. om Alt-tasten var holdt nede da klikket ble utført.

MSG

Det er definert en spesiell struct for meldingsdataene:

```
typedef struct tagMSG {
  HWND   hwnd;
  UINT   message;
  WPARAM wParam;
  LPARAM lParam;
  DWORD  time;
  POINT  pt;
} MSG, *PMSG;
```

Denne meldingspakken blir så levert til det aktuelle vinduet/ programmet ved at den blir lagt inn i en datastruktur som kalles en meldingskø.

Ethvert GUI-program som startes får umiddelbart opprettet en egen meldingskø. Ethvert GUI-program må også inneholde en hovedrutine med navnet "WinMain". Denne hovedrutinen har som sin viktigste oppgave å snappe opp meldinger fra køen, utføre en viss innledende bearbeiding av meldingen og så ekspedere den videre i en litt modifisert form.

Alle Windows-vinduer har en assosiert programsnutt som kalles "vindusfunksjonen"/"windows procedure". Vindus-funksjonen er selve overleveringsmekanismen for Windows-meldinger. Såsnart operativsystemet vet hvilket vindu det ble klikket over, så kan det slå opp i sine tabeller over registrerte vinduer og finne adressen til det aktuelle vinduets vindus-funksjon. Så kaller operativsystemet denne funksjonen med meldingspakken som parameter og dermed er meldingen

overlevert til det programmet som videre skal reagere på meldingen.

Arkitekturen til et grafisk Windowsprogram

Oppbygningen av et Windowsprogram med grafisk brukergrensesnitt er tett knyttet til meldingssystemet. Det har en arkitektur som er tilpasset mottak og håndtering av Windowsmeldinger. Og meldingene signaliserer "hendelser" inn til programmet – hendelser som programmet skal reagere på. Vi kaller dette en arkitektur for hendelsesdrevet programvare.

I et tradisjonelt C++-program (CUI-prosess) er det funksjonen main() som representerer startpunktet for eksekveringen av programmet. Alle C++-program (og alle konsoll-programmer i Windows) må ha en main()-funksjon. For GUI-programmene i Windows heter imidlertid hovedfunksjonen "WinMain()" og WinMain() har normalt et nokså standardisert innehold. I tillegg til WinMain må det også finnes en vindusfunksjon for hovedvinduet i programmet. Disse to programkomponentene samspiller med operativsystemet i et fast mønster.

I tillegg til litt oppstartskode inneholder WinMain det som kalles "meldingspumpen" – en programsløyfe der alle meldinger til vedkommende program leses inn fra en meldingskø og sendes videre til vindusfunksjonen for det aktuelle vinduet. Først i vindusfunksjonen reagerer programmet på den spesielle meldingen.

Noen viktige meldinger

WM_PAINT – sendes når innholdet i et skjermvindu skal oppdateres, tegnes opp pånytt.

WM_KEYUP – sendes når en tast er sluppet opp etter å ha vært trykket ned.

WM_LBUTTONDOWN – signaliserer at venstre musknapp er trykket ned.

"Hallo Windows" - vårt første GUI-program

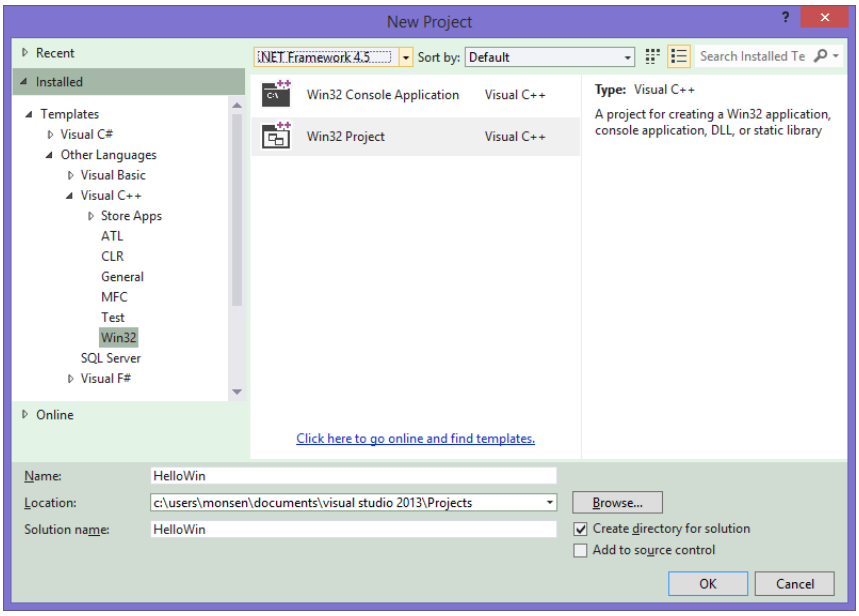
Utvikling av GUI-programmer for Windows omfatter mange detaljer og kompleksiteter – delvis på grunn av at hendelsesdrevne programmer med meldingshåndtering i utgangspunktet er relativt komplisert, men også på grunn av at vi møter nye systembibliotek med nye headerfiler, datatyper og funksjoner. Videre må vi gjøre oss kjent med en rekke nye operasjoner i selve programmeringsverktøyet Visual Studio (for at vi til slutt skal kunne arbeide enklere igjen).

For å minimalisere kompleksitet og for å gi en best mulig bakgrunnsforståelse, skal vi arbeide enkelt og primitivt med dette første programeksemplet - samtidig som vi prøver å kommentere alle nye elementer som dukker opp her. Lengere ut i kurset skal vi bruke mekanismer i Visual Studio og i programbibliotekene som gjør at vi arbeider raskere og mer effektivt, men som også skjuler mye av det vi egentlig arbeider med.

Oppretting av et prosjekt i Visual Studio

Vi skal først benytte de mekanismene i Visual Studio som betegnes AppWizard til å opprette et programmeringsprosjekt der det ligger informasjoner om hvilken programtype som skal produseres, hvilke bibliotekmoduler det skal lenkes med, samt en rekke andre "administrative detaljer" for prosjektet.

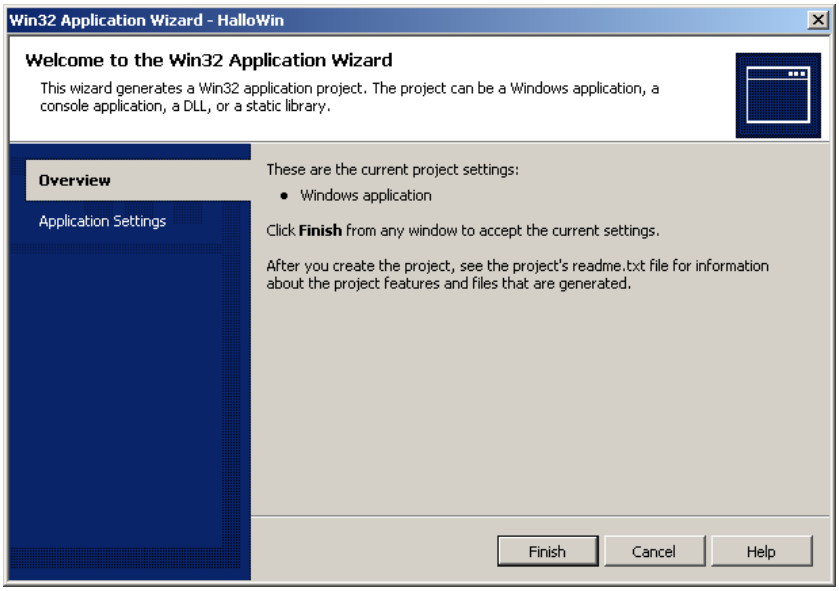
Start Visual Studio og velg File/New/Project fra menyen. Du får da opp følgende dialog der vi har åpnet mappen for C++-prosjekter og markert alternativet ”Win32 Project”.



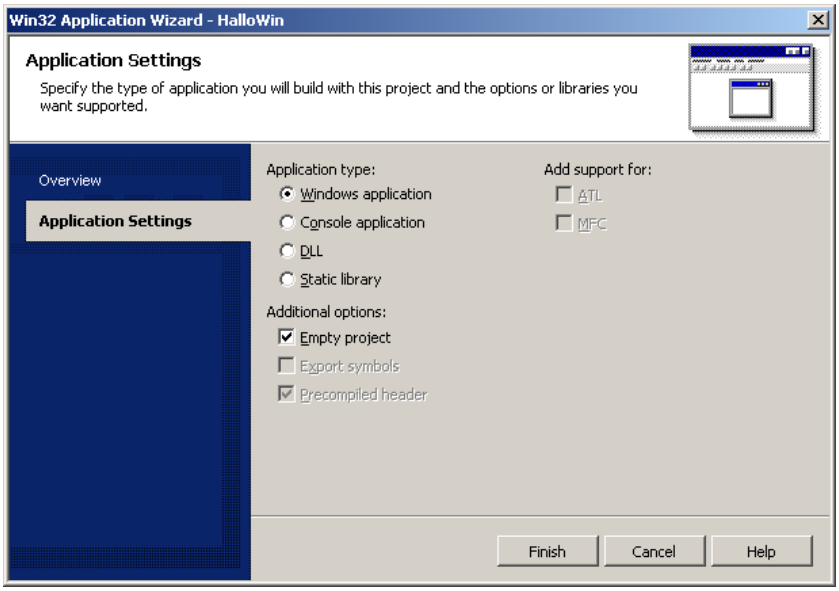
Som du ser finnes det en rekke ulike prosjekttyper som kan velges, så det er svært viktig å markere den riktige før man går videre. Lengere ut i kurset vil vi også benytte MFC Application som prosjekttyper for GUI-programmer.

Markér *Win32 Project*. Sørg deretter for at 'Location' inneholder navnet på katalogen der prosjektet skal lagres og fyll ut 'Name' med et navn for programmet. Vi skal bruke navnet HalloWin for dette eksemplet. Klikk OK.

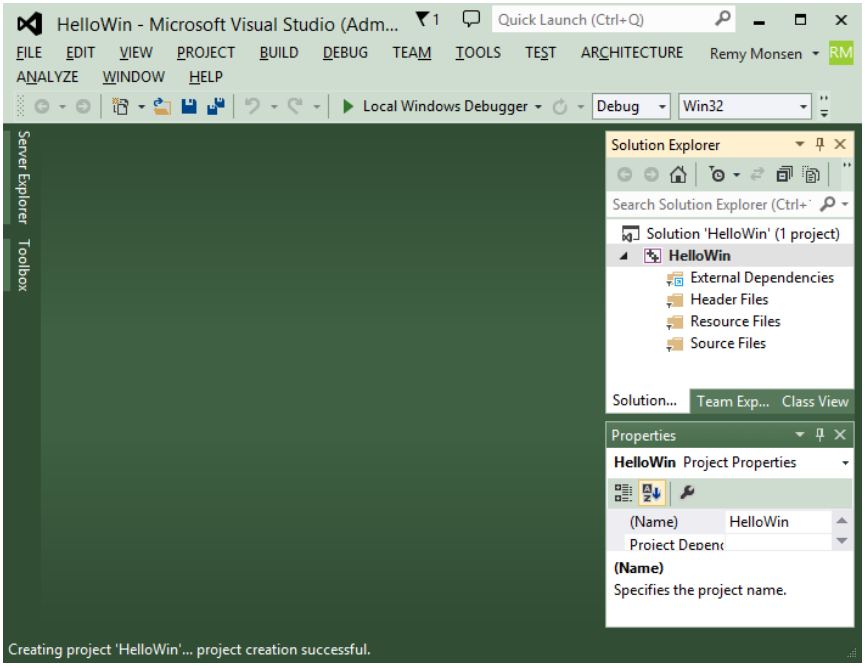
Neste dialog ser da slik ut:



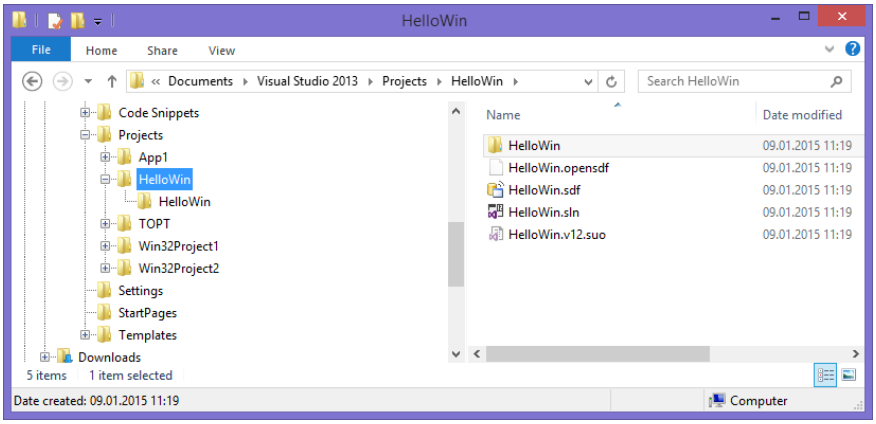
Klikk på 'Application Settings' her og markér 'Empty Project' i den neste dialogen.



Klikk på *Finish*. Visual Studio genererer så prosjektet og åpner den normale arbeidsflaten.



At Visual Studio har opprettet et prosjekt inebærer at det er laget endel ”administrative” filer i prosjektkatalogen:



Dette administrative systemet er egentlig beregnet på store og sammensatte programsystem der det endelige programmet kan være satt sammen av komponenter fra flere prosjekter og der hvert prosjekt kan omfatte mange kildefiler. Vi må likevel arbeide inne i dette systemet.

Det viktigste å merke seg er at den såkalte ”Solution-filen” (her HalloWin.sln) spiller en nøkkelrolle. Det er den som inneholder alle informasjoner om det eller de prosjektene som vi arbeider

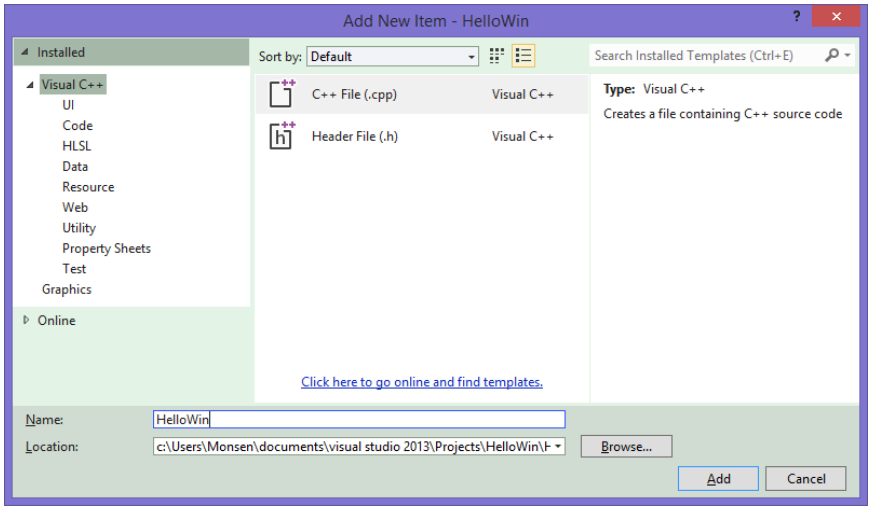
med under ett. Vi kan for eksempel dobbelklikke på en sln-fil for å starte Visual Studio med en bestemt ”solution”.

Vi har bedt om å få et tomt prosjekt, så det er ikke laget noen kildefiler for prosjektet. (Det er ’Solution View’ som viser kildefilene i prosjektet.) Derimot er det produsert fire "hushold-ningsfiler": HalloWin.sln, HalloWin.vcproj, HalloWin.ncb og HalloWin.suo. Vi skal ikke befatte oss med innholdet i disse filene, men det er viktig å være oppmerksom på at alle prosjekter i Visual Studio inneholder slike administrative filer.

Kildekode inn i prosjektet

Da er prosjektet opprettet rent administrativt. Det neste som må gjøres er å få opprettet en fil for programkoden (kildefil).

Høyreklikk på Source Files i Solution Explorer vinduet, og velg Add\New Item… og marker "C++ File (.cpp)" som filtype.



Skriv inn et filnavn (som du velger) – her "HalloWin". Vi trenger ikke å skrive inn siste del av filnavnet: ".cpp" – dette blir tilføyet automatisk. Normalt gjør vi ingen endringer på den foreslåtte plasseringen (Locatioav filen. Klikk på "Open". Da opprettes en tom fil klar for skriving av programteksten.

Nedenfor følger programteksten for HalloWin med kommentarer. Når den er skrevet eller kopiert inn kan programmet bygges og prøvekjøres.

HalloWin – kildekode

```
#include "windows.h"      // Predefinerte datatyper, makroer og
// funksjonsprototyper for bibliotekfilene i // Windows

// C RunTime Header Files
#include <tchar.h>

// Prototypdefinisjon av vindusfunksjonen
LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, UINT wParam,
        LONG lParam);

// Hovedfunksjonen og startpunktet for ethvert GUI-program
// Parametrene kommenteres i teksten etter programmet.
// ................................................
int APIENTRY _tWinMain(_In_ HINSTANCE hInstance,
        _In_opt_ HINSTANCE hPrevInstance,
        _In_ LPTSTR    lpCmdLine,
        _In_ int       nCmdShow)
{
        static TCHAR szAppName[] = _T("HalloWin");
        HWND        hwnd;// Hendel for programmets vindu
        MSG         msg; // Datatypen MSG er en struct som inneholder
        // data om en Windows-melding.
        WNDCLASS    wc;  // Datatypen WNDCLASS er ikke noen C++-klasse,
                         // men et sett av attributter som karakteriserer
                         // et vindu.
                         // Datatypen er en struct.

        // Det skal registreres en vindustype for programmets hovedvindu
        wc.style = 0;                    // standardegenskaper
        wc.lpfnWndProc = WndProc;        // Her angis hvilken
        // vindusfunksjon som
        // styrer hovedvinduet.
        wc.cbClsExtra = 0;                       // Uten kommentar
        wc.cbWndExtra = 0;                       // Uten kommentar
        wc.hInstance = hInstance;        // Programmets instans

        // Programmets ikon og cursor lastes med
        // standard ikon og pil-markør.
        wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
        wc.hCursor = LoadCursor(NULL, IDC_ARROW);

        // Vinduets bakgrunnsfarge
        wc.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);

        wc.lpszMenuName = NULL;          // Ingen meny
        wc.lpszClassName = szAppName;    // Vindus-klassen skal ha
        // et navn

        // Her kalles en systemfunksjon som registrerer informasjonene
        // i wc-structen (se ovenfor) internt i operativsystemet.
        RegisterClass(&wc);

        // Så er det klart for å opprette vinduet for programmet
        // Først opprettes vinduet som internt objekt:
        hwnd = CreateWindow(
                szAppName,       // Navnet som vindus klassen ble
                // registrert med
                _T("Titteltekst ..."),    // Teksten i tittellinjen
                WS_OVERLAPPEDWINDOW,  // En 'kode' for vinduets
                // 'stil' eller utseende
                50,      // X-posisjon for øvre venstre hjørne
                50,      // Y-posisjon for øvre venstre hjørne
                400,     // Vinduets bredde
                300,     // Vinduets høyde
                NULL,    // Uten kommentar (parent window handle)
                NULL,    // Uten kommentar (window menu handle)
```

```
                hInstance,        // Programmets instans-hendel
                NULL);   // Uten kommentar (creation parameters)

        // Deretter skal vinduet vises
        ShowWindow(hwnd, nCmdShow);

        // og tegnes opp
        UpdateWindow(hwnd);

        // Så går vi inn i 'meldingspumpen' og går i denne sløyfen helt
        // til programmet avsluttes.
        // GetMessage er en systemfunksjon som henter en og en melding
        // fra programmets meldingskø.
        while (GetMessage(&msg, NULL, 0, 0))
        {
                TranslateMessage(&msg);   // Systemfunksjon som
                // omformer visse meldinger
                DispatchMessage(&msg);    // Leverer meldingen videre
                // slik at den havner i
                // vindusfunksjonen for det
                // aktuelle vinduet.
        }
        return msg.wParam;
}


// Vindusfunksjonen / -prosedyre
LRESULT CALLBACK WndProc(HWND hwnd, UINT msg,
        UINT wParam, LONG lParam)
{
        PAINTSTRUCT ps;  // En struct som vi må ha for håndtering
                         // av WM_PAINT, men som vi ikke trenger å
                         // vite noe mer om.
        HDC hdc;         // En hendel for en 'Device Context'. Mer
                         // om denne seinere.

        switch (msg)
        {
        case WM_PAINT:
                hdc = BeginPaint(hwnd, &ps);
                TextOut(hdc, 150, 75, _T("Hallo i vinduet!"), 16);
                EndPaint(hwnd, &ps);
                return 0;

        case WM_DESTROY:
                PostQuitMessage(0);
                break;

        default:         // Hvis vi ikke håndterer meldingen selv
                         // går den til 'defaultfunksjonen'
                return  DefWindowProc(hwnd, msg, wParam, lParam);
        }

        return 0;        // Meldingen er håndtert
}
```

En gjennomgang av kildekoden i HalloWin.cpp

Hovedrutinen - WinMain

Programmet starter når operativsystemets 'Loader' har lastet opp maskinkoden og kaller WinMain:

```
int APIENTRY _tWinMain(_In_ HINSTANCE hInstance,
        _In_opt_ HINSTANCE hPrevInstance,
        _In_ LPTSTR    lpCmdLine,
        _In_ int       nCmdShow)
```

Instansparametrene hInstance og hPrevInstance er delvis foreldet og brukes ikke lenger til det de opprinnelig var definert for, nemlig å kunne sjekke om det var flere instanser av programmet igang i maskinen.

lpCmdLine er en peker til en tekststreng som inneholder eventuelle kommandoer som er gitt til programmet idet det startes.

nCmdShow er en kode som bestemmer hvordan programmets vindu skal vise. Det finnes mange alternativer:

- normalt
- usynlig
- full skjerm (maksimalisert)
- minimalisert
- . . .

Du kan teste ulike muligheter ved å tilføye f.eks.

```
nCmdShow = SW_MAXIMIZE; // eller SW_MINIMIZE
```

i starten av WinMain.

Hovedrutinen WinMain inneholder følgende lokale variabler:

```
static TCHAR szAppName[] = _T("HalloWin");
HWND        hwnd; // Hendel for programmets vindu
MSG         msg;  // Datatypen MSG er en struct som inneholder
                  // data om en Windows-melding.
WNDCLASS    wc;   // Datatypen WNDCLASS er ikke noen C++-klasse, men et
                  // sett av attributter som karakteriserer et vindu.
                  // Datatypen er en struct.
```

som kommentarene redegjør for. Merk her at _T() er en makro som gjør vanlige tekstrenger om til unicode-tekststrenger.

Den første operasjonen som utføres er et kall til RegisterClass – riktignok betinget av at hPrevInstance ikke er forskjellig fra null (noe som aldri forekommer lenger). RegisterClass tar med

seg en utfylt WNDCLASS struct – wc - som parameter. Der har vi angitt endel egenskaper vi ønsker for programmets hovedvindu, blandt annet hvilket ikon, pekersymbol eller hvilken bakgrunnsfarge vinduet skal ha:

```
wc.hIcon          = LoadIcon( NULL, IDI_APPLICATION );
wc.hCursor        = LoadCursor( NULL, IDC_ARROW );
wc.hbrBackground  = (HBRUSH) GetStockObject( WHITE_BRUSH );
```

Dessuten blir adressen til vinduets vindusfunksjon registrert:

```
wc.lpfnWndProc   = WndProc;
```

Deretter følger et kall til CreateWindow som får opprettet et vindusobjekt og der det er spesifisert flere av vinduets egenskaper.

```
hwnd = CreateWindow(
               szAppName,      // Navnet som vindusklassen ble
                               // registrert med
               _T("Titteltekst ..."), // Teksten i tittellinjen
               WS_OVERLAPPEDWINDOW,   // En 'kode' for vinduets
                                      // 'stil' eller utseende
               50,       // X-posisjon for øvre venstre hjørne
               50,       // Y-posisjon for øvre venstre hjørne
               400,      // Vinduets bredde
               300,      // Vinduets høyde
               NULL,     // Uten kommentar (parent window handle)
               NULL,     // Uten kommentar (window menu handle)
               hInstance,      // Programmets instans-hendel
               NULL  ); // Uten kommentar (creation parameters)
```

CreateWindow returnerer en vindushendel (hwnd) som vil identifisere det nye vindusobjektet. Dersom det er noe helt galt med parametrene til CreateWindow, vil det returneres NULL.

Selv om det interne vindusobjektet nå er opprettet, kommer det ikke uten videre opp noe synlig vindu på skjermen. Et vindusobjekt kan godt eksistere uten å ha noe skjermbilde (strengt tatt er det synlige skjermbildet et helt selvstendig og annet objekt enn det interne vindusobjektet). Det må kalles ytterligere to API-funksjoner før vi får fram programmets vindu på skjermen:

```
// Deretter skal vinduet vises
ShowWindow( hwnd, nCmdShow );

// og tegnes opp
UpdateWindow( hwnd );
```

Da er programmet ferdig med sine innledende operasjoner og WinMain går inn i den såkalt "meldingspumpen" der programmet vil gå inntil det termineres.

```
// GetMessage er en systemfunksjon som henter en og en melding
// fra programmets meldingskø.
while( GetMessage( &msg, NULL, 0, 0 ) )
{
   TranslateMessage( &msg );  // Systemfunksjon som
                              // omformer visse meldinger
   DispatchMessage( &msg );          // Leverer meldingen videre
                              // slik at den havner i
                              // vindusfunksjonen for det
                              // aktuelle vinduet.
}
```

GetMessage kalles med adressen til MSG-stukturen msg som parameter og returnerer ikke før det kommer en melding til programmets meldingskø. Nå skjer dette relativt raskt, blandt annet på grunn av de foregående kallene til CreateWindow som resulterer i en WM_CREATE og UpdateWindow som trigger en WM_PAINT-melding.

Returverdien fra GetMessage – som styrer while-setningen i meldingspumpen – er TRUE for alle meldinger med ett unntak: WM_QUIT. Når den meldingen mottas bryter programforløpet ut av meldingspumpen og programmet returnerer fra WinMain – dvs. at det termineres.

Vi skal ikke gå nærmere inn på hva som utføres av funksjonene TranslateMessage og DispatchMessage bortsett fra å slå fast at det er gjennom DispatchMessage at meldingene formidles videre via operativsystemet til vindusfunksjonen WndProc og ikke gjennom noe ordinært funksjonskall fra WinMain.

Vindusfunksjonen – WndProc

Vindusfunksjonen er et eksempel på noe som kalles en "Callback-funksjon" – en teknikk som blir brukt i mange sammenhenger i Windows. Uten å gå nærmere inn på detaljer omkring denne teknikken skal vi bare konstatere at funksjonen blir kalt hver gang det er hentet en melding fra meldingskøen. Det som kanskje virker uvant for oss, er at det ikke er vår kode som kaller funksjonen – og funksjonsadressen sendes heller ikke inn via DispatchMessage. Forklaringen ligger i at vi en gang for alle har informert operativsystemet om denne adressen – da vi utførte RegisterClass innledningsvis. Dermed kan operativsystemet kalle vindusfunksjonen når som helst, og vår

vindusfunksjon må være klar til å bli kalt til enhver tid. (Her er vi ved et kjerneaspekt ved "hendelsedrevet" programmering.)

Vindusfunksjonen blir kalt med parametrene hwnd, msg, wParam og lParam. hwnd er hendelen for det aktuelle vinduet, men msg er ikke lenger en full MSG-struktur som i WinMain, den er bare et heltall som representerer meldingens unike nummer (definert med symboler i winuser.h). Noen av informasjonene som fulgte meldingen gjennom WinMain, er skrellet av på veien. wParam og lParam inneholder imidlertid to av de andre elementene fra MSG-strukturen.
Den logiske strukturen for enhver vindusfunksjon vil være en switch-setning på meldingsnummeret, med case-greiner for alle meldinger man vil behandle spesielt. Alle øvrige meldinger rutes videre til en standard vindusfunksjon i operativsystemet. Vår kode får altså førsteretten til å reagere på en hendelse – for eksempel et museklikk, og hvis vi bestemmer at "that's it" – dette var alt som skulle skje, så returnerer vi 0 fra vindusfunksjonen. Da er meldingshåndteringen avsluttet. Men vi kan altså alternativt velge å sende meldingen videre ved å kalle standardfunksjonen:

```
DefWindowProc( hwnd, msg, wParam, lParam )
```

og returnere det resultatet fra denne.

Vi har nå beskrevet hovedlogikken i vindusprosedyren, men dermed har vi bare skrapet forsiktig på overflaten av det å lage et Windowsprogram. Det som gir programmene funksjonalitet, det som får programmene til å gjøre det vi ønsker at de skal gjøre, er jo innholdet i vindusfunksjonens switch-blokk. Hvilke meldinger håndterer vi og hva gjør programmet når de ulike meldingene mottas. Det er her vi må ta fatt på det omfattende arbeidet med å lære mange av de øvrige mekanismene som vi må gjøre bruk av for å kunne gi programmene våre funksjonalitet: utskrifts- og tegnefunksjoner, innputtfunksjoner via muspeker og tastatur, lagringsfunksjoner, etc.

I dette første programeksemplet er den funksjonaliteten vi har tilført programmet, begrenset til å skrive teksten "Hallo i vinduet!" på skjermen. Koden som besørger dette er:

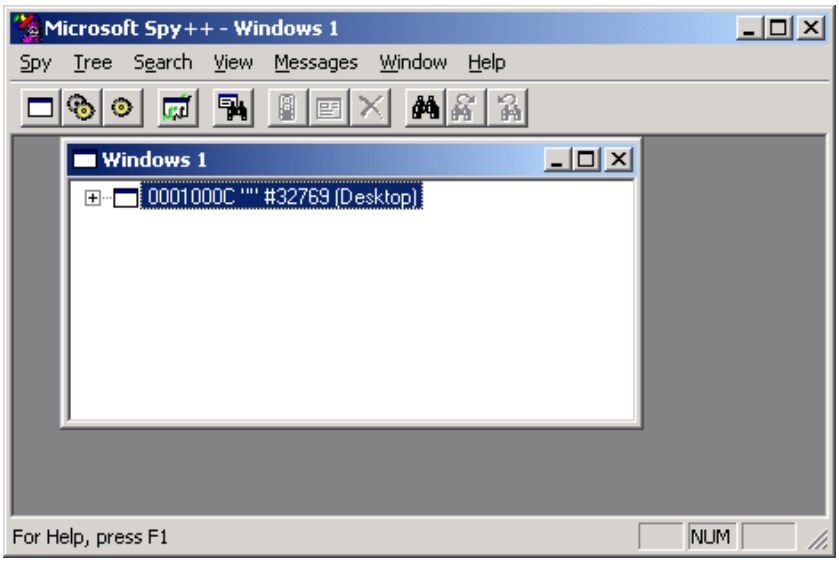
```
case WM_PAINT:
   hdc = BeginPaint( hwnd, &ps );
   TextOut( hdc, 150, 75, _T("Hallo i vinduet!"), 16 );
   EndPaint( hwnd, &ps );
   return 0;
```

Detaljene her skal vi komme nærmere inn på i avsnittet om utskriftsoperasjoner lenger ut i teksten.

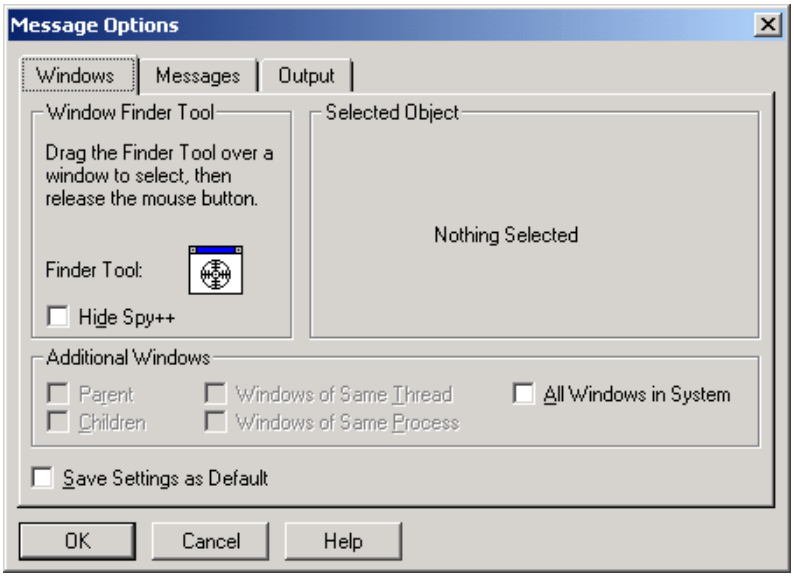
Under overflaten - SPY++

Sammen med Visual C++ følger en rekke hjelpeprogrammer. Ett av disse er Spy++ som kan brukes til å titte bak kulissene hos Windows. Det kan blant annet brukes til å spionere på vinduene og meldingene de mottar.

Start opp Spy++ fra startmenyen. Det ligger i samme meny-gruppe som Visual C++ - under Microsoft Visual Studio 6.0 Tools.

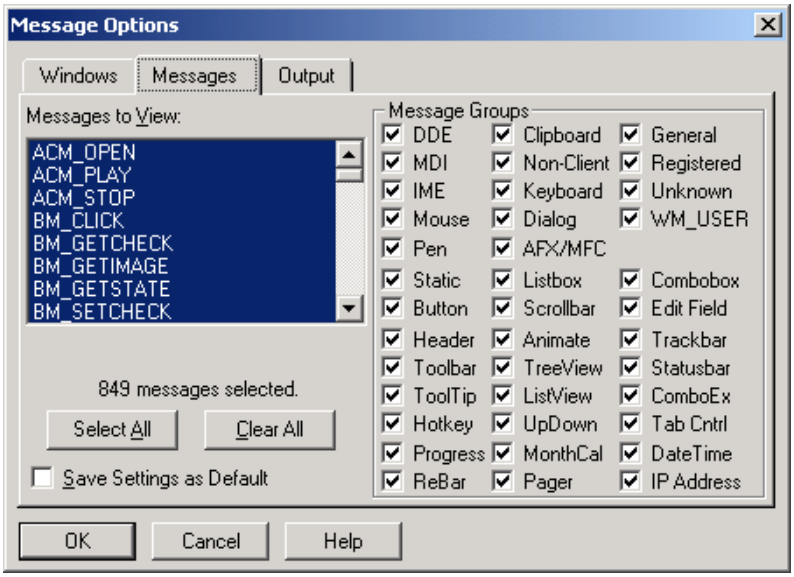


Gå til *Spy*-menyen og velg *Messages* – da åpnes en konfigur-asjonsdialog der man først skal identifisere det vinduet man vil spionere på.

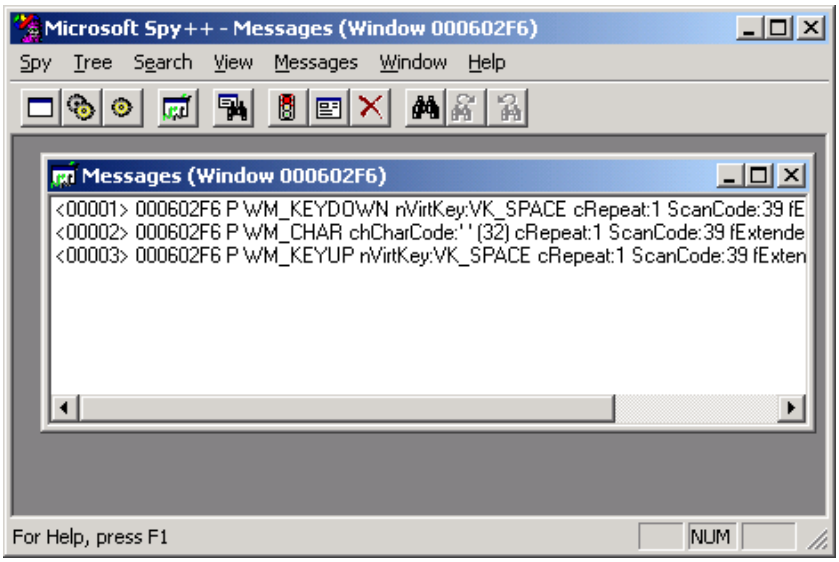


Dra ikonet som kalles *Finder Tool* over det vinduet du vil se meldingene til og slipp det der.

Klikk så på *Messages*-fanen og marker hvilke meldingstyper du vil registrere. I utgangspunktet er alle markert. Klikk først *Clear All* og deretter sjekkboksen for *Keyboard* hvis du vil logge tastaturmeldinger.



Klikk OK, aktiviser vinduet du spionerer på, bruk tastaturet, og se hvordan meldingene logges.



Spy++ har mange muligheter som kan hjelpe til med å forstå hvordan Windows opererer. Bruk informasjonen under *Help* og eksperimentér videre på egen hånd.

Øvingsoppgave 1

Målet for arbeidet med oppgaven er å få etablert en god forståelse for "vindusmeldinger" og "vindusfunksjoner" i et enkelt Windowsprogram.

Opprett et nytt prosjekt av type Win32 Application i Visual C++. Gi prosjektet et navn – for eksempel WinProg1. Velg alternativet "A Simple Win32 application".

Med filen HalloWin.cpp som mønster skal du utvide filen WinProg1.cpp slik at den får en komplett WinMain-funksjon og en vindusprosedyre som håndterer følgende meldinger:

WM_PAINT	som i HalloWin
WM_DESTROY	-------- " --------
WM_LBUTTONDOWN	
WM_RBUTTONDOWN	

WM_CHAR
WM_KEYDOWN

For de "nye" meldingene skal programmet skal vindusfunksjonen reagere med å vise en "MessageBox" med passende tekst – f.eks.

```
....
case WM_KEYDOWN:
     MessageBox( hwnd, _T("Melding: WM_KEYDOWN"),
                       _T("Info fra WinProg1"), MB_OK );
     break;
....
```

1. Bygg og prøvekjør programmet. Undersøk spesielt forholdet mellom meldingene WM_CHAR og WM_KEYDOWN.

Du skal nå utvide teksten i meldingsboksene med mer av informasjonene som følger meldingen inn i vindusfunksjonen.

```
....
case WM_CHAR:
     TCHAR MeldInfo[50];
     wsprintf( MeldInfo, _T("WM_CHAR - nr: %X, WPARAM: %X,
                         LPARAM: %X"), msg, wParam, lParam );
     MessageBox( hwnd, MeldInfo, _T("Info fra WinProg1"),
                                                    MB_OK );
     break;
....
```

Dette vil vise meldingsnummer samt de to parameterverdiene for meldingen.

Utskriftsoperasjoner – med tekst og grafikk

De datamaskinene vi er mest fortrolige med for tiden, er fortsatt svært preget av sine forfedre – de er primært regnemaskiner, tallbehandlere. Deretter er de godt utrustet for manipulering av data generelt – flytting, sortering, lagring, … De er i relativt liten grad spesialiserte for håndtering av grafiske data og for utskrift til grafiske enheter: skjermer, skrivere, plottere, etc.

Noe av de viktigste nyvinningene for "standard kontormaskiner" og personlige datamaskiner de siste 10-15 årene er at grafiske brukergrensesnitt er blitt normalt og har fått utviklet støtte i operativsystemene. Utskrift til grafiske enheter er egentlig et svært komplekst område. Enhetene varierer enormt i form, pixel-oppløsning, pixel-form, fargestøtte, støtte for skrifttyper, støtte for "sider" i et dokument, etc. Men likevel bør en tegning eller en tekstside se noenlunde lik ut enten den vises på en skjerm med en pixelflate på 800x600 eller den kommer ut på en stående A4-side fra en skriver med 300 dpi[1] pixel-oppløsning. Dette ville være en meget betydelig program-meringsoppgave hvis ikke Windows og andre operativsystem nå hadde "abstrahert" slike grafiske enheter slik at vi langt på vei slipper å tenke på hva slags enhet vi sender utskrift til. I Windows ligger denne abstraksjonen av grafiske flater i en programmodul som betegnes GDI – Graphics Device Interface.

GDI - Graphics Device Interface

Formålet med GDI-modulen er å forenkle, generalisere og abstrahere ulike utskriftsenheters egenskaper i forhold til applikasjonsprogrammer. Istedefor å kommunisere direkte med de ulike enhetenes drivere og deres spesielle egenskaper, er det i GDI lagt tilrette for at et applikasjonsprogram kommuniserer med GDI på en standardisert, overordnet måte, og at GDI håndterer de ulike enhetenes særegenheter.

[1] "dpi" står her for "Dots per inch" – pixel per tomme.

Det finnes GDI-funksjoner for utskrift av linjer, kurver, tekst og bilder. Det kan velges ulike farger og stiltyper for utskriftselementene – som "penner" av ulike tykkelser, "pensler" for ulike fyll i figurer, og skrifttyper av ulike former og størrelser.

Device Context (utskriftskontekst)

En Device Context (DC) (forsøksvis oversatt til "utskriftskontekst") er en helt sentral datastruktur i GDI – det er den som abstraherer en utskriftsenhet – en skjerm eller papirflate. Det er gjennom en slik utskriftskontekst at man oppnår "utstyrsuavhengig" grafisk utskrift fra applikasjonsprogrammenes side.

I HalloWin-eksemplet i forrige kapittel brukte vi en utskriftskontekst til å skrive ut tekst:

```
PAINTSTRUCT ps;
HDC hdc;

.....
case WM_PAINT:
   hdc = BeginPaint(hwnd, &ps);
   TextOut(hdc, 150, 75, _T("Hallo i vinduet!"), 16);
   EndPaint(hwnd, &ps);
   return 0;
.....
```

Her er **`hdc`** en hendel til et DC-objekt og **`TextOut`** er en GDI-funksjon for plassering og utskrift av tekst. Og her er det viktig å merke seg at denne koden opererer like godt mot skjermen som mot en skriver. **`hdc`**-variabelen vet ingenting om hva slags utskriftsenhet den er knyttet opp mot.

Fra dokumentasjonen:

A *device context* is a structure that defines a set of graphic objects and their associated attributes, as well as the graphic modes that affect output. The *graphic objects* include a pen for line drawing, a brush for painting and filling, a bitmap for copying or scrolling parts of the screen, a palette for defining the set of available colors, a region for clipping and other operations, and a path for painting and drawing operations. The remainder of this section is divided into the following three areas.

Det finnes flere måter å få tilgang til en utskriftskontekst. En av måtene er vist ovenfor, og brukes når utskriften er knyttet til behandlingen av WM_PAINT-meldingen. Det er på dette stedet i programmet man vanligvis utfører mesteparten av utskriftsoperasjonene. Her vil GDI-funksjonen BeginPaint returnere en hendel til en Device Context.

Dersom applikasjonen også skal skrive noe i andre deler av programstrukturen (altså utenom WM_PAINT-håndteringen), kan man skaffe seg en DC-hendel ved bruk av andre GDI-funksjoner som `CreateDC`, `GetDC`[2] og flere.

Informasjoner om utstyrsegenskaper

En Device Context vil som oftest representere en fysisk utstyrsenhet som en skjerm eller skriver. Det finnes en API-funksjon (`GetDeviceCaps`) som kan brukes for å sjekke hvilke spesifikke egenskaper den aktuelle enheten har – for eksempel hvilke oppløsning og størrelse enhetens utskriftsflate har, eller hvilke farger den støtter.

Funksjonen er deklarert som følger:

```
int GetDeviceCaps( HDC hdc, int nIndex );
```

der `hdc` er en hendel til utstyrskonteksten og `nIndex` er en av flere predefinerte indekser for hva slags egenskap man vil sjekke. Hvis man for eksempel vil undersøke pixel-oppløsningen for enheten, kaller man funksjonen to ganger – én gang for horisontal oppløsning og én gang for vertikal.

```
// bredden av utskriftsflaten
int XOppl = GetDeviceCaps( hdc, HORZRES );

// høyden av utskriftsflaten
int YOppl = GetDeviceCaps( hdc, VERTRES );
```

Tabellen nedenfor viser et utvalg av indeksene som støttes av `GetDeviceCaps`.

[2] Hjelpesystemet i Visual C++ og http://msdn.microsoft.com/library/ inneholder detaljert dokumentasjon for disse og andre API-funksjoner.

Index	Meaning
HORZSIZE	Width, in millimeters, of the physical screen.
VERTSIZE	Height, in millimeters, of the physical screen.
HORZRES	Width, in pixels, of the screen.
VERTRES	Height, in raster lines, of the screen.
LOGPIXELSX	Number of pixels per logical inch along the screen width. In a system with multiple display monitors, this value is the same for all monitors.
LOGPIXELSY	Number of pixels per logical inch along the screen height. In a system with multiple display monitors, this value is the same for all monitors.
BITSPIXEL	Number of adjacent color bits for each pixel.
PLANES	Number of color planes.
NUMBRUSHES	Number of device-specific brushes.
NUMPENS	Number of device-specific pens.
NUMFONTS	Number of device-specific fonts.
NUMCOLORS	Number of entries in the device's color table, if the device has a color depth of no more than 8 bits per pixel. For devices with greater color depths, – 1 is returned.
SIZEPALETTE	Number of entries in the system palette. This index is valid only if the device driver sets the RC_PALETTE bit in the RASTERCAPS index and is available only if the driver is compatible with 16-bit Windows.
COLORRES	Actual color resolution of the device, in bits per pixel. This index is valid only if the device driver sets the RC_PALETTE bit in the RASTERCAPS index and is available only if the driver is compatible with 16-bit Windows.
PHYSICALWIDTH	For printing devices: the width of the physical page, in device units. For example, a printer set to print at 600 dpi on 8.5-x11-inch paper has a physical width value of 5100 device units. Note that the physical page is almost always greater than the printable area of the page, and never smaller.
PHYSICALHEIGHT	For printing devices: the height of the physical page, in device units. For example, a printer set to print at 600 dpi on 8.5-by-11-inch paper has a physical height value of 6600 device units. Note that the physical page is almost always greater than the printable area of the page, and never smaller.
PHYSICALOFFSETX	For printing devices: the distance from the left edge of the physical page to the left edge of the printable area, in device units. For example, a

	printer set to print at 600 dpi on 8.5-by-11-inch paper, that cannot print on the leftmost 0.25-inch of paper, has a horizontal physical offset of 150 device units.
PHYSICALOFFSETY	For printing devices: the distance from the top edge of the physical page to the top edge of the printable area, in device units. For example, a printer set to print at 600 dpi on 8.5-by-11-inch paper, that cannot print on the topmost 0.5-inch of paper, has a vertical physical offset of 300 device units.
SCALINGFACTORX	Scaling factor for the x-axis of the printer.
SCALINGFACTORY	Scaling factor for the y-axis of the printer.

GDI-objekter og Device Context

En utskriftskontekst inneholder alltid sju GDI-objekter av typer som vist i tabellen nedenfor.

GDI-objekt	**Attributter**
Bitmap	Størrelse (i Bytes), dimensjoner (i pixler), fargedybde, komprimering, m.m.
Brush	Stil, farge, mønster
Palette	Farger og størrelse (antall farger)
Font	Skrifttype, bredde, høyde, tegnsett, m.m.
Path	Form
Pen	Stil, farge, tykkelse
Region	Plassering og dimensjoner

Når en ny utskriftskontekst opprettes blir disse objektene satt til standardverdier – eksempelvis blir penn-objektet satt til svart, tykkelse på 1 pixel, sammenhengende strek.

For å erstatte standardobjektene med andre finnes en funksjon – `SelectObject` – som "setter inn" et nytt GDI-objekt i en Device Context. Dermed blir et normalt forløp der man ønsker å bruke andre grafiske attributter enn standardverdiene, at man oppretter egne instanser av GDI-objekter og utfører `SelectObject` før man bruker en eller flere av utskrifts-funksjonene.

Utskrift knyttet til WM_PAINT

I den følgende diskusjonen vil vi snakke spesielt om utskrift til skjermen. Men det meste av det som følger vil gjelde like mye for utskriften som havner på papir fra en skriver.

Meldingen WM_PAINT sendes til et program når det skal oppdatere skjermbildet sitt, for eksempel når operativsystemet registrerer at et vindu som har vært skjult av andre vinduer, skal opp i forgrunnen. Det betyr at håndtering av WM_PAINT (koden under `case WM_PAINT:`) til enhver tid skal kunne reprodusere hele det grafiske innholdet av vinduet.

Eksemplet nedenfor er utsnitt av kode der det tegnes en rød sirkel.

```
PAINTSTRUCT ps;
HDC hdc;
HBRUSH hPensel;
HGDIOBJ hOrgPensel;

.. ..
case WM_PAINT:
   hdc = BeginPaint(hWnd, &ps);

   // GDI logikk der hdc brukes
   // Lager et pensel-objekt med rød farge
   hBrush = CreateSolidBrush(RGB(255,0,0));

   // SelectObject returnerer "det gamle objektet"
   hOrg = SelectObject(hdc, hBrush);

   // Rød sirkel
   Ellipse(hdc, 100,100, 200,200);

   // Setter tilbake original pensel
   // Særdeles viktig siden hBrush er lokal og "dør"
   // når funksjonen forlates
   SelectObject(hdc, hOrg);

   // Frigjør den lokale penselen
   DeleteObject( hBrush );

   EndPaint(hWnd, &ps);
   break;
.. ..
```

For det første vil vindusprosedyren alltid inneholde de lokale variablene

```
PAINTSTRUCT ps;
HDC hdc;
```

PAINTSTRUCT er en struct med et innhold vi ikke trenger å kjenne nærmere til nå. Den brukes som en slags hjelpevariabel for standardfunksjonene `BeginPaint` og `EndPaint`. Disse to

funksjonene skal alltid starte og avslutte koden for WM_PAINT-håndteringen.

hPensel og hOrgPensel brukes begge som hendler til pensel-objekter. hPensel er av typen HBRUSH mens hOrgPensel er av en mer generell type for GDI-objekter: HGDIOBJ.

Mellom `BeginPaint` og `EndPaint` utføres utskriftsoperasjonene samt den tilhørende administrasjonen av GDI-objekter:

- opprett GDI-objekt (det kan være opprettet på forhånd)
- gjør det nye objektet aktivt i utskriftskonteksten ved kall til SelectObject (den returnerer det "gamle" objektet)
- utfør en eller flere utskriftsoperasjoner ved kall til tegne- og/eller tekstfunksjoner i GDI (her bruker vi funksjonen Ellipse(….)
- sett tilbake de gamle GDI-objektene (de nye forsvinner hvis de er opprettet lokalt)
- frigi ressurser i nye GDI-objekter (hvis de er opprettet lokalt) ved å kalle DeleteObject.

Uansett hvor omfattende og komplisert koden under WM_PAINT blir, vil den grovt sett operere på denne måten.

De viktigste typene av GDI-objekter

Brush-objekter

En *brush/pensel* er et grafisk virkemiddel som bestemmer farge og utseende for den innvendige bakgrunnen i lukkede figurer som rektangler, polygoner, sirkler.

Det finnes to typer pensler – logiske og fysiske. En logisk pensel er en beskrivelse av en ideell bitmap som skal brukes til utskriften. En fysisk pensel er en virkelig bitmap som kan brukes på det aktuelle utstyret. Når en applikasjon oppretter et pensel-objekt, returneres en hendel til en logisk pensel. Når denne hendelen sendes som parameter til SelectObject, vil device-driveren opprette en fysisk pensel basert på denne logiske penselen. Den fysiske penselen vil ligge så nær opp til den logiske penselen som mulig.

Pen-objekter

Penn-objekter brukes til å produsere linjer og kurver. Det er sju attributter som definerer en penn: tykkelse, "stil", farge, mønster, skravering, endetype og "join style".

Font-objekter

Font-objekter er selvsagt knyttet til utskrift av tekst. En font – eller "skriftsnitt" som er en mer korrekt betegnelse – er en samling av tegn og symboler som deler et felles design. De tre viktigste elementene av en font-design er form, stil og størrelse.

Størrelsen av en font er ikke helt presist definert, men normalt vil det være avstanden fra bunnen av en liten g til toppen av en stor M. Størrelsen angis vanligvis i en måleenhet som kalles "punkt" – en tradisjonell typografisk enhet. Et punkt er definert som 1/72 av en tomme.

GDI-Demo – et litt større eksempel

Den følgende teksten er et litt mer omfattende eksempel på bruk av GDI-operasjoner og utskriftskonteksten. I dette kompendiet vil vi ikke gå noe særlig mer inn på de enkelte GDI-funksjonene. De er for det meste relativt enkle å ta ibruk og er beskrevet grundig i online-dokumentasjon for Visual C++.

```
#include "windows.h"

#include <tchar.h>

LRESULT CALLBACK WndProc(HWND hwnd, UINT msg,
        UINT wParam, LONG lParam);

int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
        LPSTR lpCmdLine, int nCmdShow)
{
        static TCHAR szAppName[] = _T("GDIDemo");
        HWND        hwnd;
        MSG         msg;
        WNDCLASS    wc;

        if (!hPrevInstance)
        {
                wc.style = CS_HREDRAW | CS_VREDRAW;
                wc.lpfnWndProc = WndProc;
                wc.cbClsExtra = 0;
                wc.cbWndExtra = 0;
                wc.hInstance = hInstance;
                wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
                wc.hCursor = LoadCursor(NULL, IDC_ARROW);
                wc.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
                wc.lpszMenuName = NULL;
                wc.lpszClassName = szAppName;

                RegisterClass(&wc);
        }
```

```
        hwnd = CreateWindow(
                szAppName,       // den registrerte vindusklassen
                _T("Demonstrasjon av enkelte GDI-funksjoner"),
                                                 // vindustittel
                WS_OVERLAPPEDWINDOW, // vindusstil
                CW_USEDEFAULT,   // x-posisjon for øvre venstre hjørne
                CW_USEDEFAULT,   // y-posisjon for øvre venstre hjørne
                400,             // bredde i pixler
                500,             // høyde i pixler
                NULL,            // ingen foreldrevindu
                NULL,            // ingen meny
                hInstance,       // instanshendel
                NULL);           / ingen oppstartsparametre

        ShowWindow(hwnd, nCmdShow);
        UpdateWindow(hwnd);

        while (GetMessage(&msg, NULL, 0, 0))
        {
                TranslateMessage(&msg);
                DispatchMessage(&msg);
        }

        return msg.wParam;
}


// Vindusfunksjonen / -prosedyren
LRESULT CALLBACK WndProc(HWND hwnd, UINT msg,
        UINT wParam, LONG lParam)
{
        HDC hdc;                          // hendel til device context
        PAINTSTRUCT ps;                   // hjelpevariabel
        POINT ptForrige;                  // hjelpevariabel
        HPEN hOrgPenn, hMinPenn;  // hendler til to penn-objekter
        HBRUSH hOrgPensel, hMinPensel;    // hendler til to pensel-
                                          // objekter
        HFONT hOrgFont, hMinFont; // hendler til to font-objekter

        switch (msg)
        {
        case WM_PAINT:
                hdc = BeginPaint(hwnd, &ps);      // standard åpning

                // lager et nytt penn-objekt, rød med prikk-strek format
                hMinPenn = ::CreatePen(PS_DASHDOT, 1, RGB(255, 0, 0));

                // setter inn den nye pennen - den gamle tas vare på
                hOrgPenn = (HPEN) ::SelectObject(hdc, hMinPenn);

                // starter fra (50,50), tegner en strek
                ::MoveToEx(hdc, 50, 50, &ptForrige);
                ::LineTo(hdc, 250, 250);

                // bytter tilbake til den gamle pennen og tegner en
                // strek til
                ::SelectObject(hdc, hOrgPenn);
                ::LineTo(hdc, 50, 250);

                // lager en ny pensel - blå og tett
                hMinPensel = ::CreateSolidBrush(RGB(0, 0, 255));

                // setter inn den nye penselen - den gamle tas vare på
                hOrgPensel = (HBRUSH) ::SelectObject(hdc, hMinPensel);

                // tegner et kvadrat
                Rectangle(hdc, 100, 100, 200, 200);

                // lager en ny font - en serie med parametre, men de
                // fleste kan settes til "default" - det er bare første
                // og siste parameter som er "ikke-default"
                hMinFont = ::CreateFont(36, 0, 0, 0, FW_MEDIUM, FALSE,
                        FALSE, FALSE, DEFAULT_CHARSET,
                        OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS,
                        DEFAULT_QUALITY, DEFAULT_PITCH,
                        _T("Times New Roman"));
```

```
                // setter inn den nye fonten - den gamle tas vare på
                hOrgFont = (HFONT) ::SelectObject(hdc, hMinFont);

                // skriver litt tekst
                TextOut(hdc, 50, 300, _T("Her er min nye font."), 20);

                // bytter til den gamle fonten og skriver litt til
                ::SelectObject(hdc, hOrgFont);
                TextOut(hdc, 50, 400,
                        _T("... og er er standardfonten."), 28);

                // rydder opp
                ::DeleteObject(hMinPenn);
                ::DeleteObject(hMinPensel);
                ::DeleteObject(hMinFont);

                EndPaint(hwnd, &ps);      // standard avslutning

                return 0;

        case WM_DESTROY:
                PostQuitMessage(0);
                break;

        default:   // la andre meldinger få "standardbehandling"
                return  DefWindowProc(hwnd, msg, wParam, lParam);
        }

        return 0;
}
```

Ferdiglagde GDI-objekter

Under initialiseringen av GDI-modulen opprettes det et sett av predefinerte, standard GDI-objekter som kan benyttes av alle applikasjoner. Disse objektene kalles "stock objects" – lagrede objekter. Med unntak for regioner og bitmap-er, har hver objekttype minst ett lagret objekt.

Det gir raskere kode og lavere minneforbruk å benytte lagrede objekter enn å opprette egne objekter. Funksjonen `GetStockObject` returnerer en hendel til et lagret objekt – for eksempel et standard penn-objekt. Deretter brukes denne hendelen på samme måte som andre objekthendler. Man skal imidlertid ikke utføre `DeleteObject` for lagrede objekter.

```
HGDIOBJ GetStockObject( int fnObject );
```

fnObject kan være en av følgende verdier:

Value	**Meaning**
BLACK_BRUSH	Black brush.
DKGRAY_BRUSH	Dark gray brush.
DC_BRUSH	**Windows 2000/XP:** Solid color brush. The default color is white. The color can be changed by using the **SetDCBrushColor** function.

GRAY_BRUSH	Gray brush.
HOLLOW_BRUSH	Hollow brush (equivalent to NULL_BRUSH).
LTGRAY_BRUSH	Light gray brush.
NULL_BRUSH	Null brush (equivalent to HOLLOW_BRUSH).
WHITE_BRUSH	White brush.
BLACK_PEN	Black pen.
DC_PEN	**Windows 2000/XP:** Solid pen color. The default color is white. The color can be changed by using the **SetDCPenColor** function.
WHITE_PEN	White pen.
ANSI_FIXED_FONT	Windows fixed-pitch (monospace) system font.
ANSI_VAR_FONT	Windows variable-pitch (proportional space) system font.
DEVICE_DEFAULT_FONT	**Windows NT/2000/XP:** Device-dependent font.
DEFAULT_GUI_FONT	Default font for user interface objects such as menus and dialog boxes. This is MS Sans Serif.
OEM_FIXED_FONT	Original equipment manufacturer (OEM) dependent fixed-pitch (monospace) font.
SYSTEM_FONT	System font. By default, the system uses the system font to draw menus, dialog box controls, and text.
SYSTEM_FIXED_FONT	Fixed-pitch (monospace) system font.
DEFAULT_PALETTE	Default palette. This palette consists of the static colors in the system palette.

Koordinatsystemer og "mapping mode"

Inntil nå har vi operert i et "standard" koordinatsystem for utskriftsflaten, der origo ligger i øvre venstre hjørne av vinduets klientområde, x-aksen går mot høyre og y-aksen peker nedover. Måleenheten i dette koordinatsystemet er pixsler.

Dette er default koordinatsystem og er enkelt å bruke til enkle grafiske oppgaver. Men det finnes en rekke andre mulige koordinatsystem for en Device Context. Ett av attributtene til en Device Context kalles "Mapping Mode" – det er denne (sammen med enkelte andre) som bestemmer hvilken type koordinatsystem utskriftsflaten vil ha.

Det er definert åtte ulike "mapping modes" for Windows:

Mapping Mode	**Logisk enhet**	**Stigende x-verdier**	**Stigende y-verdier**
MM_TEXT	pixel	Mot høyre	Nedover
MM_LOMETRIC	0,1 mm	Mot høyre	Oppover
MM_HIMETRIC	0,01 mm	Mot høyre	Oppover
MM_LOENGLISH	0,01 in.	Mot høyre	Oppover
MM_HIENGLISH	0,001 in.	Mot høyre	Oppover
MM_TWIPS[3]	1/1440 in.	Mot høyre	Oppover
MM_ISOTROPIC	Fri (x = y)	Fri	Fri
MM_ANISOTROPIC	Fri (x != y)	Fri	Fri

En "logisk enhet" er måleenheten for et koordinatsystem i en Device Context. Ved utskrift til en fysisk utskriftsflate må de logiske koordinatverdiene regnes om til "fysiske enheter". Generelt vil de aller fleste GDI-funksjonene med koordinater som parametre, ta koordinatverdiene i logiske enheter.

[3] Twip er et kunstig ord med betydningen "twentieth of a point" – et tjuendedels punkt. Et punkt regnes som 1/72 tomme, dermed blir en twip lik 1/1440 tomme.

Uten at begrepet "logisk enhet" har vært nevnt tidligere, er det slike enheter vi har benyttet i eksemplene til nå – for eksempel i: LineTo(hdc, 250, 50);
der x-verdien angis til 250 logiske enheter og y-verdien til 50. Når vi også har referert til disse enhetene som pixler skyldes det at vi hittil bare har operert i default mapping mode, som er MM_TEXT.

Det har imidlertid store fordeler å kunne frigjøre seg fra pixel-enheter – som er svært utstyrsavhengige – og til eksempelvis å kunne benytte 0,1 mm (MM_LOWMETRIC). Den største fordelen ligger i at samme kode vil gi samme utskriftsstørrelser både på skjerm og på andre enheter, for eksempel en papir-utskrift. Med pixel-baserte enheter ville man måtte bruke ulike koordinatverdier for ulike utskriftsenheter.

Vi bruker API-funksjonen SetMapMode for å endre Mapping Mode for en Device Context:

```
int SetMapMode( HDC hdc, int fnMapMode );
```

Her er hdc en konteksthendel og fnMapMode et av de predefinerte uttrykkene for Mapping Mode i tabellen ovenfor.

Alle andre Mapping Modes enn MM_TEXT har y-verdiene som stigende oppover, noe som er vanligst. Men origo ligger fortsatt i øvre venstre hjørne, så y-verdiene vil være negative, noe som ikke er helt optimalt. Det finnes GDI-funksjoner som kan flytte origo, men vi går ikke inn på disse i denne sammenhengen.

Øvingsoppgave 2

(Bruk av enkle tegnefunksjoner)

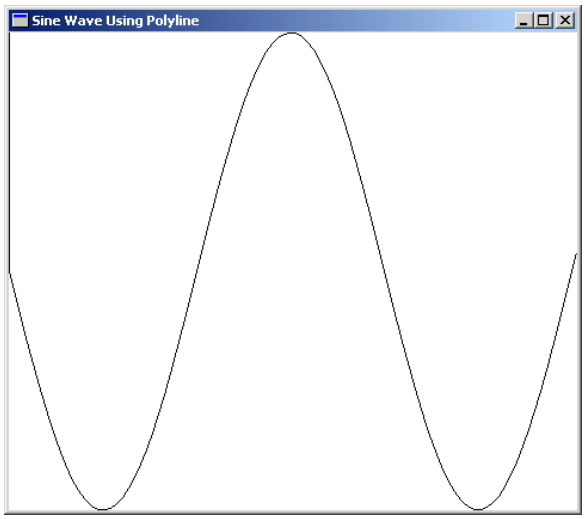
Opprett et nytt prosjekt av type Win32 Application i Visual C++ på ditt arbeidsområde. Gi prosjektet navnet Sinus. Velg alternativet "An empty project".

Opprett en fil – Sinus.cpp – i prosjektkatalogen din. Innholdet skal være som vist nedenfor.

Utfør meny-valget "Project / Add to project / Files ..." og velg Sinus.cpp inn i prosjektet. Bygg og prøvekjør.

Studér programkoden i Sinus.cpp og test med debugger/ breakpoint.
- Hva er funksjonen til meldingen WM_SIZE?
- Hva slags størrelser representerer cxClient og cyClient?

Kan du lage tegnemekanismen mer avansert slik at sinuskurven til enhver tid "fyller ut vinduet" som vist nedenfor?



Kode-eksempel – Sinus.cpp

```
#include <windows.h>
#include <math.h>
#include <tchar.h>

LRESULT CALLBACK WndProc( HWND hwnd, UINT msg,
                          UINT wParam, LONG lParam ) ;


//////////////////////////////////////////////////////////////
int APIENTRY WinMain( HINSTANCE hInstance,
                      HINSTANCE hPrevInstance,
                          PSTR szCmdLine, int iCmdShow)
{
   static TCHAR szAppName[] = _T("Sinuskurve");
   HWND     hwnd ;
   MSG      msg ;
   WNDCLASS  wc ;

   wc.style         = CS_HREDRAW | CS_VREDRAW;
   wc.lpfnWndProc   = WndProc;
```

```
   wc.cbClsExtra    = 0;
   wc.cbWndExtra    = 0;
   wc.hInstance     = hInstance;
   wc.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
   wc.hCursor       = LoadCursor (NULL, IDC_ARROW);
   wc.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
   wc.lpszMenuName  = NULL;
   wc.lpszClassName = szAppName;

   RegisterClass (&wc);

   hwnd = CreateWindow (  szAppName,
                          _T("Sinuskurve av linjestykker"),
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL);

   ShowWindow (hwnd, iCmdShow);
   UpdateWindow (hwnd);

   while (GetMessage (&msg, NULL, 0, 0))
   {
      TranslateMessage (&msg);
      DispatchMessage (&msg);
   }
   return msg.wParam;
}


LRESULT CALLBACK WndProc( HWND hwnd, UINT iMsg,
                       WPARAM wParam, LPARAM lParam)
{
   static int  cxClient, cyClient ;
   HDC         hdc ;
   int         i ;
   PAINTSTRUCT ps ;

   switch (iMsg)
   {
      case WM_SIZE:
         cxClient = LOWORD (lParam) ;
         cyClient = HIWORD (lParam) ;
         return 0 ;

      case WM_PAINT:
         hdc = BeginPaint (hwnd, &ps) ;

         for( i=0; i<100; i++ )
            LineTo( hdc, i*5, 200 + sin( 0.1 * i ) * 200 );

         return 0 ;

      case WM_DESTROY:
         PostQuitMessage (0) ;
         return 0 ;
   }

   return DefWindowProc( hwnd, iMsg, wParam, lParam) ;
}
```

Håndtering av bruker-innputt

Mus-hendelser

Windows genererer meldinger når musmarkøren beveges over et vindu og når det klikkes på en av musknappene. Meldingene går til det vinduet som er synlig umiddelbart under musmarkøren – uansett om dette vinduet er det aktive vinduet eller ikke (med unntak for visse situasjoner).

De meldingene som er spesielt aktuelle for applikasjonsprogrammer, er vist i tabellen nedenfor (for mus med to knapper). Dette er de meldingene som genereres over vinduenes klientområder. Det finnes andre meldinger som genereres over "non-client" områder, men de blir normalt håndtert av standard vindusfunksjon (DefWindowProc).

Melding	**Hendelse**
WM_MOUSEMOVE	Musmarkør flyttes en viss lengde
WM_LBUTTONDOWN	Venstre musknapp trykket ned
WM_LBUTTONUP	Venstre musknapp sluppet opp
WM_LBUTTONDBLCLK	Venstre musknapp trykket ned for andre gang på kort tid
WM_RBUTTONDOWN	Høyre musknapp trykket ned
WM_RBUTTONUP	Høyre musknapp sluppet opp
WM_RBUTTONDBLCLK	Høyre musknapp trykket ned for andre gang på kort tid

For alle disse meldingene inneholder lParam skjermkoordinatene for musmarkøren da hendelsen inntraff. x-koordinaten ligger i de laveste 16 bit av lParam og y-koordinaten i de 16 høyeste. Du kan bruke makroene LOWORD og HIWORD for å trekke ut disse verdiene på en lettvint måte:

```
x = LOWORD( lParam );
y = HIWORD( lParam );
```

I wParam ligger det et bit-mønster som indikerer status for både musknapper og Shift- og Ctrl-tastene. Se online-dokumentasjonen for flere detaljer.

Kodefragmentet nedenfor viser en håndtering av WM_RBUTTONDOWN-melding

```
case WM_RBUTTONUP:
   int x, y;
   TCHAR s[256];

   x = LOWORD(lParam);
   y = HIWORD(lParam);
   wsprintf(s, _T("Høyre knapp trykket i posisjon: (%d,
                                          %d)."),x, y);
   if (wParam & MK_CONTROL)
      wcscat_s(s, _T("\nKontrolltasten var også trykket."));

   ::MessageBox(hwnd, s, _T(""), MB_OK);
   return 0;
```

NB! Strengfunksjonen `wcscat_s` *krever at header-filen "tchar.h" blir inkludert.*

Andre meldinger trigget av musklikk

Det er ikke bare meldinger i tabellen ovenfor som kan bli sendt på grunn av et musklikk. Når muspekeren brukes for å klikke på en menylinje, vil det også bli sendt en WM_COMMAND-melding. Det samme vil skje om menykommandoen velges ved hjelp av tastaturet. Vi vil komme tilbake til den viktige WM_COMMAND-meldingen lengre ut i teksten.

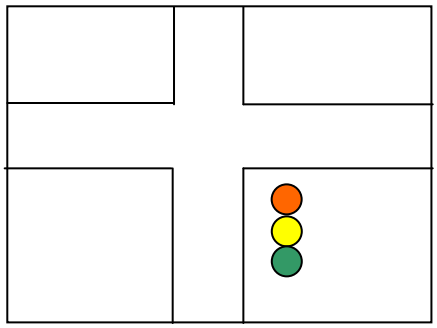
Øvingsoppgave 3

Del 1:

- Start opp Microsoft Visual Studio.
- Velg File/New. Under Projects velger du Win32 Application.
- Tast inn et prosjektnavn, for eksempel Oving3. Behold standardvalgene i veiviseren.
- Det blir nå laget et prosjekt. Skift til FileView og åpne filen som inneholder WinMain(..) – Oving3.cpp
- Skriv inn eller kopier over innholdet i Oving3.cpp slik det er gjengitt nedenfor eller fra tilsvarende fil på fagets web-side. Test at alt fungerer. (Programmet skal tegne og fargelegge diverse!)

Del 2:
Les **grundig** om alle de nye API kallene etter WM_PAINT.

Del 3: Lage enkelt lyskryss.
Du skal nå lage et lyskryss som tegnes ut på skjermen. Se skisse til høyre. Trafikklyset lages slik: Farget sirkel (Lys på), vanlig sirkel (lys av). Trafikklyset skal kunne vise rødt, gult og grønt. (Figuren til høyre viser alle lysene på!)



RGB(r,g,b)-makroen angir en farge sammensatt av en rød, en grønn og en blå komponent. Andelen fra en komponent angis som et tall mellom 0 og 255. F.eks vil RGB(0,255,0) angi grønt. RGB(0,0,0) er svart og RGB(255,255,255) er hvitt.

Koden for å lage lyskryss må legges inn under håndteringen av WM_PAINT.

Lyset i trafikklyset skal skifte når du trykker ned **venstre** musknapp.

Del 4: Lage to trafikklys.
Lag to veier som krysser hverandre og plassér et trafikklys ved hver av veiene. For å gjøre det enkelt, tenker vi oss to enveiskjørte gater. Også her brukes hendelsen **venstre** musknapp for å la lysene veksle.

Innhold i Oving3.cpp – kode for start av øvingen

```
#include "stdafx.h"

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

int APIENTRY _tWinMain(_In_ HINSTANCE hInstance,
   _In_opt_ HINSTANCE hPrevInstance,
   _In_ LPTSTR    lpCmdLine,
   _In_ int       nCmdShow)
{
   WNDCLASS wc;
   TCHAR     szClass[] = _T("WND_NR_1");

   memset(&wc, 0, sizeof(wc));
```

```
   wc.style    = CS_HREDRAW | CS_VREDRAW;
   wc.lpfnWndProc    = (WNDPROC)WndProc;
   wc.cbClsExtra     = 0;
   wc.cbWndExtra     = 0;
   wc.hInstance      = hInstance;
   wc.hIcon          = 0;
   wc.hCursor        = LoadCursor(0, IDC_ARROW);
   wc.hbrBackground  = (HBRUSH)(COLOR_WINDOW+1);
   wc.lpszClassName  = szClass;

   if( RegisterClass(&wc)==0 )
      return -1;

   // Create the window
   HWND hWnd = CreateWindow(szClass, _T("Øving 3"),
               WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, 0,
                  CW_USEDEFAULT, 0, 0, 0, hInstance, 0);
   if (!hWnd) return FALSE;

   ShowWindow(hWnd, nCmdShow);
   UpdateWindow(hWnd);

   // Main message loop:
   MSG msg;
   while (GetMessage(&msg, NULL, 0, 0))
   {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
   }
   return 0;
}



LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
                              WPARAM wParam, LPARAM lParam)
{
   PAINTSTRUCT ps;
   HDC hdc;

   switch (message)
   {
      case WM_DESTROY:
         PostQuitMessage(0);
         break;

      case WM_PAINT:
         hdc = BeginPaint(hWnd, &ps);

         HBRUSH hBrush;
         HGDIOBJ hOrg;

         // Velger rød børste
         hBrush = CreateSolidBrush(RGB(255,0,0));
         hOrg = SelectObject(hdc, hBrush);
         Ellipse(hdc, 100,100, 150,150);

         // Velger gul børste
         hBrush = CreateSolidBrush(RGB(255,255,0));
         SelectObject(hdc, hBrush);
         Ellipse(hdc, 100,150, 150,200);

         // Velger grønn børste
         hBrush = CreateSolidBrush(RGB(0,255,0));
         SelectObject(hdc, hBrush);
         Ellipse(hdc, 100,200, 150,250);

         // Setter tilbake orginal børste
         SelectObject(hdc, hOrg);
```

```
         DeleteObject(hBrush);

         // Flytter posisjon og tegner en strek!
         MoveToEx(hdc, 125,250,0);
         LineTo(hdc, 125,400);

         EndPaint(hWnd, &ps);
         break;

      default:
         return DefWindowProc(hWnd, message, wParam, lParam);
       }
   return 0;
}
```

Timer-mekanismen i Windows

Windows inneholder muligheten til å få levert en form for tidsstyrt melding til et program. Meldingen det er snakk om heter WM_TIMER, og den mekanismen som vil sende meldingen kalles en "Timer". En timer er altså et Windows-objekt som periodisk sender WM_TIMER-meldingen til et gitt vindu.

Vi omtaler timere her, ikke fordi de utgjør noen grunnleggende og viktig mekanisme i Windows, men fordi de kan være nyttige hjelpefunksjoner i mange sammenhenger. De finner ofte anvendelse i spill, simuleringsprogrammer eller programmer der visse automatiske funksjoner skal utføres periodisk.

For å få opprettet og startet en timer, utføres API-funksjonen:

```
UINT SetTimer(
     HWND hWnd,              // vindushendel
     UINT nIDEvent,          // timer-id
     UINT nElapse,           // tidsrom mellom hver melding
                             // - i millisekunder
     TIMERPROC lpTimerFunc   // Callback-funksjon
   );
```

der `hWnd` er hendel en til det vinduet som skal motta meldingene. `nIDEvent` er et heltall som vil fungere som et identitetsnummer for vedkommende timer og `nElapse` er antall millisekund mellom hver melding. `lpTimerFunc` kan være en "callback-funksjon", men skal bare settes lik `NULL` i våre eksempler og oppgaver.
Funksjonen

```
BOOL KillTimer( HWND hWnd, UINT nIDEvent )
```

stopper og fjerner en timer.

```
.. ..
case WM_LBUTTONDOWN:
   SetTimer( hWnd, 1, 1000, NULL);
   return 0;

case WM_TIMER:
   g_tall = g_tall+1;
   InvalidateRect( hWnd, 0, TRUE);
   return 0;

case WM_PAINT:
   TCHAR sz[100];

   hdc = BeginPaint( hWnd, &ps );
   wsprintf( sz, _T("Tallet er %d"), g_tall );
   TextOut( hdc, 100, 100, sz, strlen(sz) );

   EndPaint( hWnd, &ps );
   return 0;

case WM_RBUTTONDOWN:
   KillTimer( hWnd, 1 );
   return 0;
.. ..
```

Det følgende eksemplet viser et kodefragment der et klikk med venstre musknapp starter en timer, der en tellervariabel økes med en for hver gang WM_TIMER-meldingen mottas, der tellerverdien skrives og oppdateres på skjermen og der et høyreklikk stopper og fjerner timeren.

Øvingsoppgave 4

Prøvekjør eksempelprogrammet "Reaksjon.exe". Dette er et enkelt program som måler tiden det tar for brukeren å reagere på en hendelse.

Vindusfunksjonen i "Reaksjon.exe" inneholder håndtering av meldingene WM_RBUTTONDOWN, WM_LBUTTONDOWN og WM_TIMER i tillegg til WM_PAINT og WM_DESTROY som før.

Du skal på grunnlag av forklaringene nedenfor, skrive et program som virker på samme måte som Reaksjon.exe.

Det bruker to integer variabler

```
int t1, t2;
```

for starttid og sluttid.

Programlogikken er i grove trekk:

- Skriv startteksten på skjermen
- Når brukeren høyreklikker startes en "timer" med tilfeldig intervall:

```
int d = rand()    // tilfeldig intervall mellom 0 og
                  // RAND_MAX (mer enn 4 millioner), bør
                  // justeres så d ikke blir for stor
SetTimer( 1, d, NULL ); // d angir timer-intervallet i
                        // millisekunder
```

- Skjermteksten endres.
- Når WM_TIMER-meldingen mottas, skiftes skjermteksten til "VENSTREKLIKK NÅ !!" samtidig som t1 settes: `t1 = clock();`
  og timeren stanses.
- Når WM_LBUTTONDOWN kommer, settes t2: `t2 = clock();` og skjermteksten endres til å vise differansen mellom t2 og t1 – millisekunder.

Demoprogrammet har ingen spesiell håndtering av muligheten for å venstreklikke **før** signalet er gitt – prøv å bygge inn funksjonalitet som fanger opp slike hendelser og som da skriver "TJUVSTART!" på skjermen og avbryter testen (ikke programmet som helhet).

Tastetrykk

Til tross for mus og grafisk brukergrensesnitt, er tastaturet fortsatt den viktigste innputt-enheten for svært mange program. Trykkes en tast ned, så genereres meldingen WM_KEYDOWN (eller WM_SYSKEYDOWN for ALT- eller F10-tasten). Når tasten slippes opp kommer WM_KEYUP eller WM_SYSKEYUP.

For alle de fire tastetrykk-meldingene inneholder lParam seks informasjonsfelt: Repeat Count (bit 0-15), OEM Scan Code (bit 16-23), Extended Key Flag (bit 24), Context Code (bit 29), Previous Key State (bit 30) og Transition State (bit 31).

Hvilket vindu får tastaturmeldingene?

Musmeldingene går (med visse unntak) til det vinduet som er synlig under musmarkøren. Men hvordan avgjøres det hvilket vindu som skal ha tastaturmeldingene? Det er to begrep som kommer inn her. For det første vil det normalt alltid være ett og bare ett vindu som "har fokus", og tastaturmeldingene går da til dette vinduet.

Men det kan forekomme situasjoner da ingen av vinduene har fokus. I så fall går meldingene til det "aktive vinduet" – som det også bare kan være ett av om gangen. Unntaket fra denne regelen gjelder dersom det aktive vinduet er minimalisert.

Repeat Count

Repeat Count (repetisjonstallet) er antall tastetrykk som meldingen representerer. Dette er vanligvis 1, men dersom en tast holdes nede lenge og vindusfunksjonen ikke svelger unna meldingene fort nok, så kan det tenkes at tallet kan være mer enn 1.

OEM Scan Code

Dette er den koden som genereres av tastatur-elektronikken og som representerer et salgs identitetsnummer for vedkommende tast. Denne koden benyttes lite i Windows siden tastene kan identifiseres på en mer hardware-uavhengig måte

Context Code, Previous Key State og Transition State

Disse bit-ene er ikke så vesentlige i vår sammenheng, så vi viser til online-dokumentasjonen for detaljer.

Virtuelle tastaturkoder

Parameteren wParam er viktigere enn innholdet i lParam, den inneholder en tallverdi som betegnes "Virtual Key Code", og som er en bedre identifikasjon av hvilken tast som er trykket enn scankoden. Dette kodesettet er definert av Microsoft for å ha en utstyrsuavhengig identifikajon av alle tastene og tastekombinasjonene som kan forekomme på PC-er.

Nedenfor er det liste litt av winuser.h der det er definert navn for de virtuelle tastaturkodene. Der ser vi at også musknapper er tatt med som "virtuelle taster".

```
// Virtual Keys, Standard Set
```

```
#define VK_LBUTTON        0x01
#define VK_RBUTTON        0x02
#define VK_CANCEL         0x03
#define VK_MBUTTON        0x04
#define VK_BACK           0x08
#define VK_TAB            0x09
#define VK_CLEAR          0x0C
#define VK_RETURN         0x0D
#define VK_SHIFT          0x10
#define VK_CONTROL        0x11
#define VK_MENU           0x12
#define VK_PAUSE          0x13
#define VK_CAPITAL        0x14
```

De virtuelle tastaturkodene for bokstavtegnene i det engelske alfabetet er identiske med ANSI-koden for vedkommende stor bokstav.

Tastene SHIFT, ALT og CTRL

Dersom vi trenger å sjekke om en av disse hjelpetastene har vært trykket samtidig med en annen tast som vi har mottatt melding for, kan vi kalle følgende API-funksjon.

```
short GetKeyState( int nVirtKey ):
```

Her kan *nVirtKey* da være: VK_SHIFT, VK_CAPIAL eller VK_CONTROL. (Det går også an å sjekke om en av musknappene har vært nede samtidig med tastetrykket. Bruk *nVirtKey* lik: VK_LUTTON, eller VK_RBUTTON.) Returverdien fra `GetKeyState` er negativ dersom den aktuelle hjelpetasten var trykket.

Kodeeksempel (utdrag fra vindusprosedyren):

```
case WM_KEYDOWN:
{
   if( GetKeyState(VK_SHIFT) < 0 )
      MessageBox(0,_T("SHIFT var trykket!"), _T(""), MB_OK);
   else
      MessageBox(0,_T(" SHIFT var ikke trykket!"),
                                          _T(""),MB_OK);

   return 0;
}
```

Andre meldinger fra tastatur-innputt

På samme måte som museklikk, resulterer tastetrykkene ofte også i andre meldinger – først og fremst WM_CHAR eller WM_SYSCHAR. Men som nevnt tidligere kan også menykom-

mandoer iverksettes ved hjelp av tastaturet og dermed utløse en WM_COMMAND.

Anta at tasten Q er trykket. Dette genererer en WM_KEYDOWN-melding med wParam lik 0x00000051 og lParam lik 0x00100001. På veien gjennom meldingspumpen blir meldingen analysert i funksjonen TranslateMessage. Der finner Windows ut at denne tasten er knyttet til et vanlig tekst-tegn: 'q' ('Q' hvis Shift var trykket), og dermed sender Windows en WM_CHAR-melding med wParam lik 0x00000071 og uendret lParam, umiddelbart etter WM_KEYDOWN og før en eventuell WM_KEYUP får slippe til. wParam inneholder da ANSI-koden for tegnet som er assosiert med den aktuelle tasten. I dette tilfellet ANSI-koden for 'q'.

Oppsummering

Piltaster og andre spesialtaster kan behandles ved å studere meldingen WM_KEYDOWN (evt WM_KEYUP). Bokstaver må behandles ved å studere WM_CHAR. WM_CHAR meldinger dannes av funksjonen TranslateMessage i meldingspumpen.

Eksemler på meldinger som genereres ved tastetrykk:

Liten E (Caps Lock av)

Melding	wParam	
WM_KEYDOWN	0x00000045	- Virtuell kode for E-tasten
WM_CHAR	0x00000065	- ANSI koden til ‘e’
WM_KEYUP	0x00000045	- Virtuell kode for E-tasten

Liten E (Caps Lock på)

Melding	wParam	
WM_KEYDOWN	0x00000045	- Virtuell kode for E-tasten
WM_CHAR	0x00000045	- ANSI koden til ‘E’
WM_KEYUP	0x00000045	- Virtuell kode for E-tasten

Stor A via Shift-tasten (Caps Lock av)

Melding	wParam	
WM_KEYDOWN	0x00000010	- Virtuell kode: VK_SHIFT
WM_KEYDOWN	0x00000041	- Virtuell kode for A-tasten

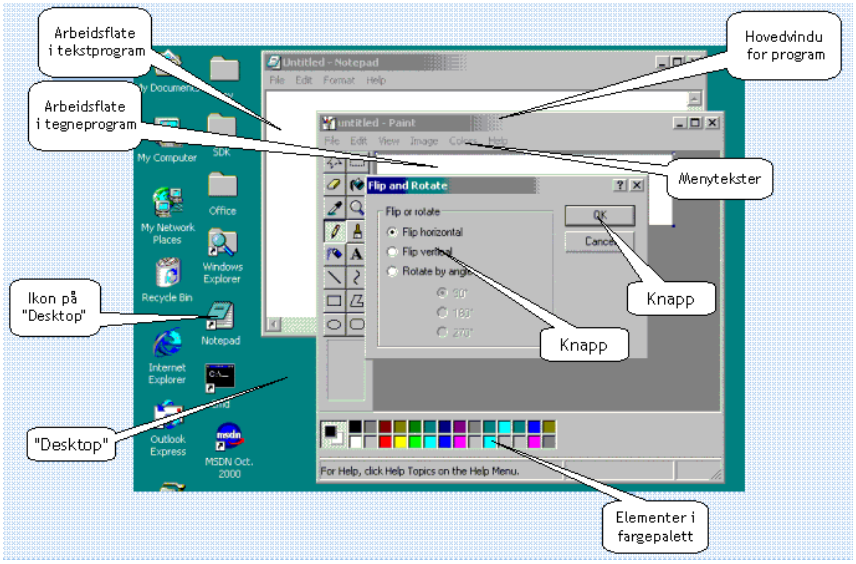
WM_CHAR	0x00000041	- ANSI koden til 'A'
WM_KEYUP	0x00000041	- Virtuell kode for A-tasten
WM_KEYUP	0x00000010	- Virtuell kode: VK_SHIFT

Visuelle elementer i grafiske brukergrensesnitt

Både i Windows og i andre grafiske brukergrensesnitt finnes det et sett av standardelementer som brukes i kommunikasjonen med brukeren. De viktigste er:

- menyer
- dialogvinduer
- knapper, lister, tekstfelt, kombibokser, m.fl. – som utgjør innholdet i dialogvinduer.

Og som vi slo fast innledningsvis: "Alt er vinduer". API-biblioteket til Windows inneholder disse vindustypene klar til bruk.



Det som skiller dem fra hverandre er delvis at de opprettes med ulike "styles", og delvis at de selvsagt har helt ulike vindusfunksjoner.

Ulike vindustyper – relasjoner mellom vinduer.

Ikke uventet kan man opprette vinduer med en rekke ulike egenskaper. Vi kan dele vindustypene inn i fire hovedkategorier som styres av dwStyle-parameteren i CreateWindow.

```
HWND CreateWindow(
  LPCTSTR lpClassName,  // klassetype (ref RegisterClass!)
  LPCTSTR lpWindowName, // vindustekst (tittel)
  DWORD dwStyle,        // vindustype, masse ulike parameter!!!
  int x,                // koordinater
  int y,                //
  int nWidth,           // størrelse
  int nHeight,          //
  HWND hWndParent,      // foreldrevindu
  HMENU hMenu,          // meny eller ID til "barn" vindu
  HINSTANCE hInstance,  // program "handle"
  LPVOID lpParam        // div.info
);
```

Overlappende (Overlapped - opprettet med WS_OVERLAPPED)

Dette er toppvinduer med tittelramme og kant. Hovedvinduet i et Windowsprogram er av denne typen. Et slikt vindu vil ha hWndParent = 0.

Eid (Owned window)

Spesialversjon av Overlapped som er eid av et annet vindu. Når eiervinduet blir minimalisert, gjøres dette usynlig. Når eier-vinduet dør, dør også dette vinduet.

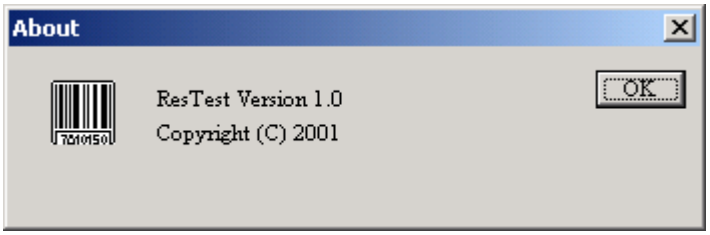
Popup-vindu (opprettet med WS_POPUP)

Er en vindustype uten tittelramme. Og det vil typisk være eid av et annet vindu. Standard menyer vil være vinduer av denne typen.

Child-vindu (opprettet med WS_CHILD)

Er alltid eid av en annet vindu, og alltid plassert innenfor foreldrevinduet. Et child-vindu "følger" alltid foreldrevinduet både med tanke på minimaliserting, flytting, endring av størrelse etc. Det er parameteren hWndParent i CreateWindow som angir foreldrevinduet når et child-vindu opprettes.

For et dialogvindu som nedenfor, er det dialogvinduet som representerer en container eller foreldrevindu, mens OK-knappen, ikon og tekstlinjene er Child-vindu i dialogen.
Child-vinduer i en dialog betegnes også som "Child Window Controls" eller "kontrollvinduer".



Kontrollvinduene i en dialog er konstruert til å inngå i et samspill med sitt containervindu. De har kun en delrolle i brukergrensesnittet, der det er dialogvinduet som helhet som skal kontrollere den overordnede logikken. Kontrollvinduene er derfor bygget slik at de informerer sitt foreldrevindu om "alle viktige hendelser" og lar foreldrevinduet bestemme hva som skal skje videre.

WM_COMMAND

Informasjonen tilbake fra kontrollvinduer i en dialog til foreldrevinduet kalles gjerne "notifications" og skjer vanligvis ved hjelp av en spesialmelding WM_COMMAND. Hvis for eksempel en knapp blir klikket i en dialog, så vil dialogvinduet motta en WM_COMMAND-melding der wParam inneholder en predefinert "notification code": BN_CLICKED og en ID for vedkommende knapp. Dermed kan vindusfunksjonen for dialogvinduet bestemme hva brukeren ønsket.

Kodeeksempel - oppretter 10 child-vinduer (av klassen "button") i et hovedvindu.

```
// Her opprettes 10 stk vinduer av type "button"
// "button" er en ferdigregistrert klasse i Windows

case WM_CREATE:
   int i;
   for( i=0; i < 10; i++ )
   {
      TCHAR sz[100];
```

```
      wsprintf(sz, _T("Knapp nr %d"), i);
      CreateWindow(  "button",
                     sz,
                     WS_CHILD|WS_VISIBLE|BS_PUSHBUTTON,
                     50,                              // xpos
                     i*25,                            // ypos
                     100,                             // dx
                     20,                              // dy
                     hWnd,                            // parent
                     (HMENU)i,                        // id
                     hInst,
                     0);
   }
   return 0;

case WM_COMMAND:     // Fanger opp nedtrykking av knapp
   int id;
   id  = LOWORD(wParam);
   if (id >= 0 || id <=9)
   {
      TCHAR sz[100];
      wsprintf(sz,_T("Du trykket knapp %d"), id);
      MessageBox(0,sz,_T(""), MB_OK);
   }
```

På lignende vis blir WM_COMMAND også benyttet av et menyvindu som blir klikket. Menyvinduet har et foreldrevindu som vil motta denne informasjonen om hvilket menyvalg som ble utført.

Noen standard kontrolltyper

Når vi skal få fram et hovedvindu for et ordinært Windows-program, utfører vi først en RegisterClass-operasjon der vi registrerer en ny vindustype spesielt for vårt program før vi kan utføre CreateWindow. Windows inneholder imidlertid en rekke forhåndsregistrerte vindustyper som er klare til bruk – blant annet endel standard kontrolltyper.

Kontrolltype	**Predefinert vindusklasse**
Trykknapp (flere subtyper)	"button"
"Read Only" tekst	"static"
Tekstvindu med redigeringsmuligheter	"edit"
"Skyve-kontroll"	"scrollbar"
Liste av tekster	"listbox"
Kombinasjon av redigeringsfelt og tekstliste	"combobox"

Vi kommenterer nå to av disse kontrolltypene litt nærmere. Se ellers til online dokumentasjon for flere detaljer.

Trykknapper

Vi har standard trykknapper med ulikt utseende og ulik oppførsel:

vanlige knapp

sjekkboks

radioknapp
m. fl.

De tilhører imidlertid den samme kontrolltypen "button", men skiller seg fra hverandre ved ulike "Windows Styles" som parametre til CreateWindow. Den vanlige knapptypen har stilen BS_PUSHBUTTON, sjekkbokser har BS_CHECKBOX, mens radioknapper har BS_RADIOBUTTON.

Kontroller av typen "button" kan sende følgende "notifications" via WM_COMMAND (ikke komplett liste):

```
BN_CLICKED
BN_DISABLE
BN_DOUBLECLICKED
```

Det er definert fem spesialmeldinger for "button"-kontroller – dvs. meldinger som kan sendes med API-funksjonen SendMessage fra foreldrevinduet til en knapp. Disse er:

```
BM_GETCHECK - Henter "sjekkstatus"
BM_SETCHECK - Setter knappen "sjekket" eller
              "usjekket"
BM_GETSTATE - Henter knappens status – "sjekket",
               "trykket" eller "udefinert"
BM_SETSTATE - Setter knappens status
BM_SETSTYLE - Endrer knappens stil
```

Tekstbokser

Standard tekstfelt der brukeren kan skrive ord og setninger som innputt til programmet, blir som oftest implementert ved et vindu av den predefinerte "edit"-typen. Ved hjelp av spesielle stil-kategorier for disse vinduene kan man automatisk få midtstilt, venstre- eller hørejustert tekst, ulike "scrolle-mekanismer" og velge mellom et vindu for en enkel tekstlinje eller flere linjer.

Som for andre kontroller, vil også edit-kontroller sende "notification"-meldinger via WM_COMMAND. De spesielle hendelseskodene er:

```
EN_SETFOCUS    - kontrollen har fått fokus
EN_KILLFOCUS   - kontrollen har mistet fokus
EN_CHANGE      - teksten skal til å endres
EN_UPDATE      - teksten er endret
EN_ERRSPACE    - kontrollen har for lite minne
EN_MAXTEXT     - kontrollen har for lite minne til
                 ny tekst
EN_HSCROLL     - horisontal "scroll"
EN_VSCROLL     - vertikal "scroll"
```

Det finnes svært mange spesialmeldinger for "edit"-kontroller. Noen av dem er: WM_CUT, WM_COPY, WM_CLEAR som vil får kontrollen til å klippe, kopiere eller bare slette tekst som er markert i tekstboksen.

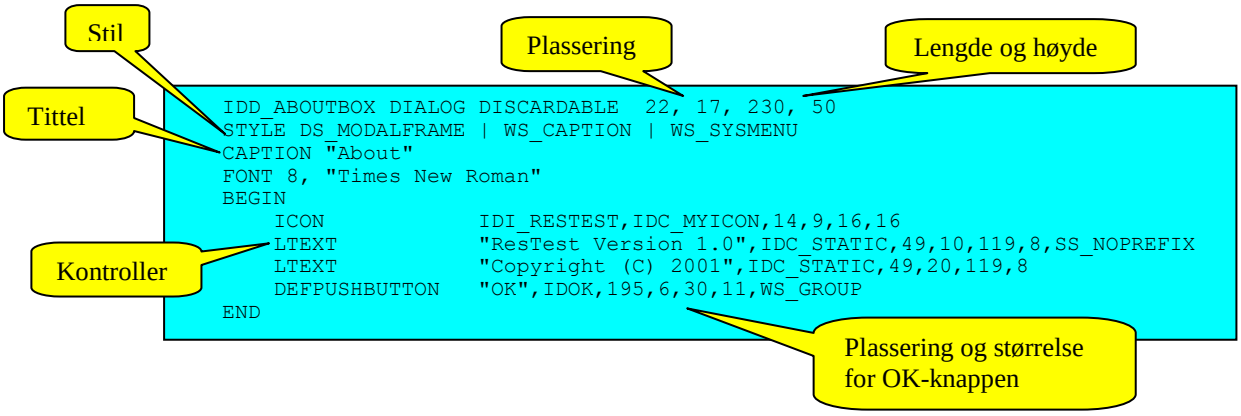
Nedenfor er videre bare listet noen få av andre spesialmeldinger for edit-kontroller. Meldingsnavnene indikerer stikkordsmessig hva de kan brukes til.

```
EM_GETLINE
EM_GETLINECOUNT
EM_GETMARGINS
EM_GETSEL
EM_LINELENGTH
EM_LINESCROLL
EM_SCROLL
EM_SETMARGINS
EM_SETREADONLY
EM_UNDO
```

Ressurser og ressurseditorer

Før vi går videre med detaljer og eksempler vedrørende bruken av WM_COMMAND skal vi imidlertid introdusere "Windows ressurser" og ressurseditorene for dialoger og menyer.

For lettere å kunne arbeide med den visuelle utformingen av brukergrensesnittene har Windows muligheter for å beskrive menyer, dialoger og visse andre elementer i tekstfiler som kalles "ressursskript". Teksten nedenfor er et utdrag av et ressursskript – det beskriver en liten dialog med to kanpper.




```
IDD_ABOUTBOX DIALOG DISCARDABLE  22, 17, 230, 50
STYLE DS_MODALFRAME | WS_CAPTION | WS_SYSMENU
CAPTION "About"
FONT 8, "Times New Roman"
BEGIN
    ICON            IDI_RESTEST,IDC_MYICON,14,9,16,16
    LTEXT           "ResTest Version 1.0",IDC_STATIC,49,10,119,8,SS_NOPREFIX
    LTEXT           "Copyright (C) 2001",IDC_STATIC,49,20,119,8
    DEFPUSHBUTTON   "OK",IDOK,195,6,30,11,WS_GROUP
END
```

På grunnlag av slike skript kan Windows automatisk generere mesteparten av den koden som skal til for å få opp den aktuelle dialogen.

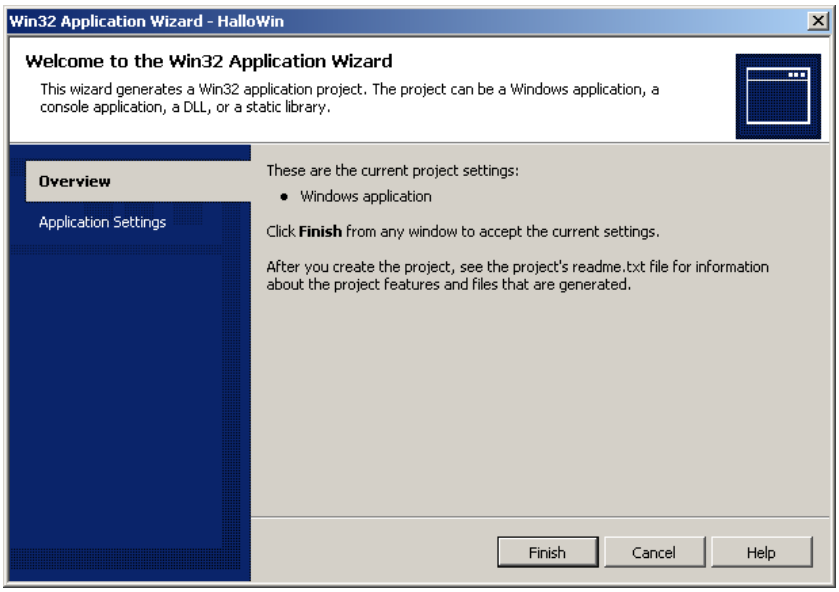
I tidlige utgaver av programmeringsverktøyene ble disse skriptfilene behandlet "manuelt" – i teksteditoren. Selv det var vesentlig mer effektivt enn å kode dialogvinduet og kontrollene i C/C++ som indikert i kodeeksemplet på forrige side. I dagens utgaver av Visual C++ behandler vi ikke lenger ressurs-skriptene på den måten. Vi har fått hjelpeprogrammer (ressurseditorer) der vi kan bygge opp både menystrukturer og dialoger visuelt. Produksjon av skriptfiler, kompilering og integrering av ressursene med den øvrige programkoden er automatisert og foregår i all hovedsak uten vår medvirkning.

Ressursskriptene er imidlertid fortsatt tilstede som en del av kildekoden for et prosjekt. Disse filene har filtypen RC.

Vi skal nå gjennomgå et eksempel der vi starter med et Win32-prosjekt der AppWizard har lagt tilrette for bruk av ressurser. Så skal vi legge inn en menykommando og deretter en dialog som skal åpnes ved hjelp av menykommandoen.

Eksempel med meny- og dialogressurs

Velg '*Close Solution*' eller start Visual Studio pånytt og velg File/New/Project samt Win32 Project som tidligere. Tast inn et prosjektnavn – for eksempel RessursTest - men velg nå å bruke standard-alternativet, altså uten å markere *'Empty Project'*. Bare klikk '*Finish*' og bygg det genererte prosjektet.



I dette tilfellet får vi generert et mer komplett program med to cpp-filer, tre h-filer og tre ressurs-filer. Når vi prøvekjører programmet slik det kommer "ut av boksen", ser vi at vi har fått en menylinje med to menytitler: File og Help – hver med én kommando under seg: Exit og About … .

Ser vi nærmere på koden i vindusfunksjonen WndProc i RessursTest.cpp, finner vi en case-gren som fanger opp WM_COMMAND og håndterer den på følgende måte:

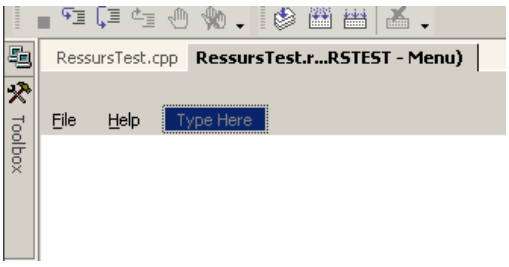
```
case WM_COMMAND:
   wmId    = LOWORD(wParam);
   wmEvent = HIWORD(wParam);  // blir ikke brukt her

   switch( wmId )    // Parse the menu selections:
   {
   case IDM_ABOUT:
      DialogBox(hInst, (LPCTSTR)IDD_ABOUTBOX,
                              hWnd, (DLGPROC)About);
      break;

   case IDM_EXIT:
      DestroyWindow(hWnd);
      break;

   default:
      return DefWindowProc(hWnd, message, wParam, lParam);
   }
   break;
```

Det er altså WM_COMMAND som bringer med seg informasjon om hvilken menykommando som blir utført. I de 16 laveste bitene av wParam ligger en meny-ID som identifiserer den aktuelle kommandoen. Hvor kommer denne ID-en fra, og hvordan er de to meny-ID-ene tildelt navnene IDM_ABOUT og IDM_EXIT. Det er selvfølgelig AppWizard som har fikset opp noe her, men hvordan kan vi får gjort noe tilsvarende?

Ny menykommando

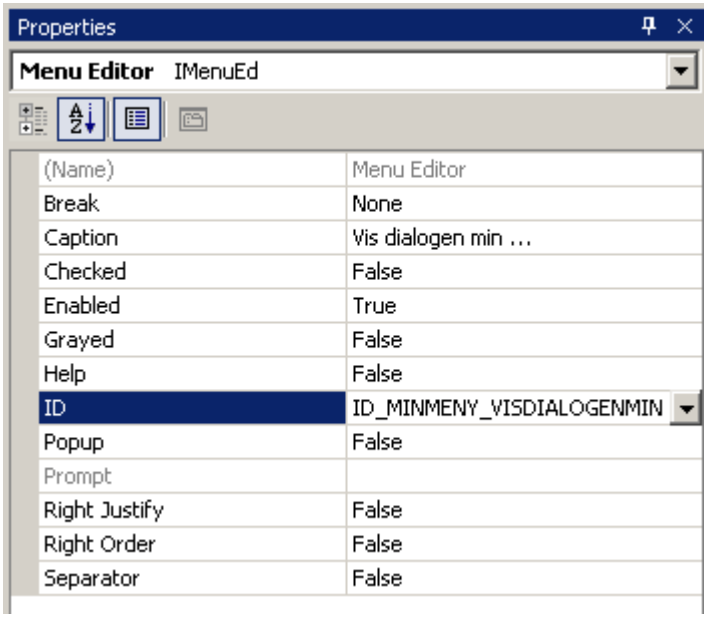
La oss legge inn en ny menytittel: "*Min meny*", og en ny kommando under den: "*Vis dialogen min ...*". Da går vi til Resource View i Solution-vinduet, åpner ressurs-treet som vist i figuren til venstre, og dobbelklikker på menyressursen: IDC_RESSURSTEST.

Vi får da opp vinduet for redigering av menyressursen. Det fylte rektanglet indikerer plassen for neste ledige menytittel, og hvis vi bare starter å skrive, blir det oppfattet som



om vi er igang med en ny meny.

Under den nye menytittelen åpnes det et felt for en menykommando. Der skriver vi "Vis dialogen min ...". Menykommandoen blir automatisk tildelt en ID som vises i Properties-vinduet.



Bak i kulissene blir det så automatisk tilordnet et heltall til dette ID-symbolet. Mer presist: det blir skrevet en #define-linje inn i filen resource.h som assosierer symbolet med et heltall (forskjellig fra andre meny-ID-er). Dermed kan vi gjenkjenne kommandoen i vindusfunksjonen ved hjelp av et tekstlig symbol og slipper å huske ID-numrene selv.

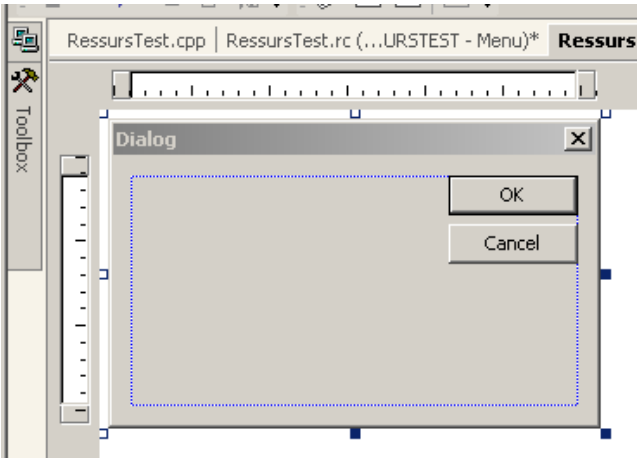
```
// fra resource.h
#define ID_MINMENY_VISDIALOGENMIN  32771
```

La oss teste den nye menykommandoen. Legg inn følgende linjer i håndteringen av WM_COMMAND – umiddelbart foran `case IDM_ABOUT:`

```
case ID_MINMENY_VISDIALOGENMIN:
   MessageBox(hWnd, _T("Menykommandoen funker !!"),
                                          _T(""), MB_OK );
   break;
```

Ny dialog

Så var det dialogen da. Lukk menyredigeringsvinduet, høyre-klikk på "Dialog" i ressurs-treet, og velg "Insert Dialog" fra popup-menyen. Det åpnes da et vindu for dialogredigering.



I Properties-vinduet kan dialogens tittel (Caption) samt en rekke andre egenskaper redigeres – for eksempel ID, font, posisjon i forhold til foreldrevinduet, m.m.

Hva skal så til for å få en dialog i sving – ved hjelp av en menykommando? Jo, menykommandoen dukker opp som en WM_COMMAND i vindusfunksjonen til hovedvinduet (WndProc). Vi ser av koden for håndtering av About-kommandoen at en dialog kan åpnes ved API-funksjonen DialogBox:

```
INT DialogBox(
  HINSTANCE hInstance,  // instanshendel
  LPCTSTR lpTemplate,   // dialog-ID casted som streng
  HWND hWndParent,      // hendel til foreldrevindu
  DLGPROC lpDialogFunc  // dialogens vindusprosedyre
);
```

Så i første omgang kan vi bare kopiere vindusprosedyren for About-dialogen og forandre navnet til TestDlgProc:

```
LRESULT CALLBACK TestDlgProc( HWND hDlg, UINT message,
                              WPARAM wParam, LPARAM lParam )
{
   switch( message )
   {
      case WM_INITDIALOG:
         return TRUE;

      case WM_COMMAND:
         switch(LOWORD(wParam))
         {
         case IDOK:
         case IDCANCEL:
            EndDialog(hDlg, LOWORD(wParam));
            return TRUE;
         }
         break;
   }
   return FALSE;
}
```

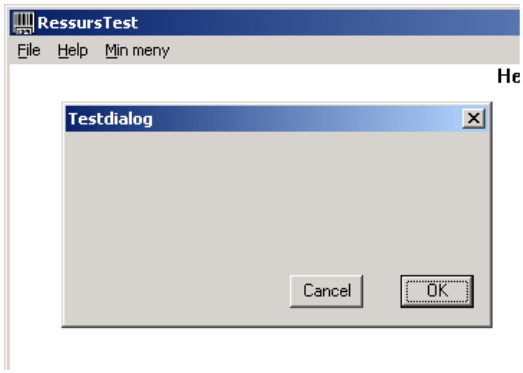
Kopier også funksjonsprototypen for About og bytt navn:

```
LRESULT CALLBACK  TestDlgProc(HWND, UINT, WPARAM, LPARAM);
```

Sett så inn et kall til DialogBox der menykommandoen `ID_MINMENY_VISDIALOGENMIN` håndteres i hovedvinduets vindus-funksjon:

```
case ID_MINMENY_VISDIALOGENMIN:
   // MessageBox(hWnd, _T("Menykommandoen funker !!"),
   //                                        _T(""), MB_OK );

   DialogBox(hInst, (LPCTSTR)IDD_DIALOG1, hWnd,
                                     (DLGPROC)TestDlgProc);
   break;
```

Bygg prosjektet, og dermed skulle det være klart til å bruke den nye menykommandoen og få fram dialogen.

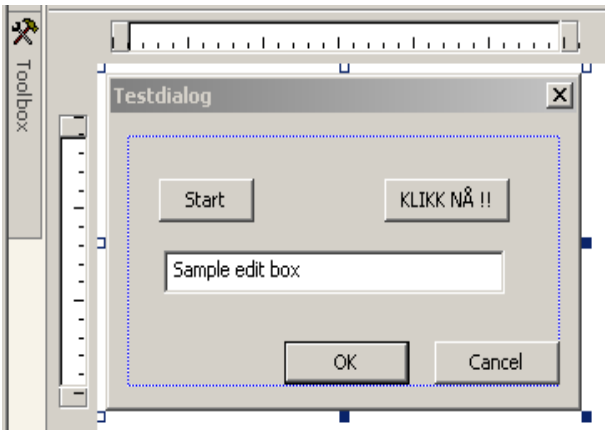


Men kan vi gjøre noe mer enn dette ut av denne dialogen?

La oss bygge en enkel reaksjonstester med dialogen som brukergrensesnitt og logikken lagt inn i dialogens vindusfunksjon.

En enkel reaksjonstester

En reaksjonstester er et program som måler reaksjonstiden til brukeren fra et signal gis og til brukeren klikker en knapp.



Gå tilbake til dialogeditoren og legg inn to knapper og en Edit-kontroll som vist i figuren til høyre.

Knappen med teksten "KLIKK NÅ !!" er gitt ID-en IDC_KLIKK, Start-knappen har ID-en IDC_START og Edit-kontrollen har ID-en IDC_TEKST. Legg merke til at for KLIKK NÅ knappen er

egenskapen "Visible" satt til False – den skal ikke være synlig med det samme dialogen vises. Nå har vi brukergrensesnittet klart.

Funksjonaliteten skal være som følger. Når brukeren klikker på Start skal det gå en tilfeldig tid, og så skal knappen KLIKK NÅ vises. Samtidig med at KLIKK NÅ vises skal programmet registrere tidspunktet – starttidspunktet for reaksjonstiden. Når brukeren klikker på KLIKK NÅ knappen, registreres tiden pånytt – slutt-tidspunktet for reaksjonstiden. Reaksjonstiden blir differensen mellom det siste og det første tidspunktet.

Vi bruker en timer til å gi oss pausen mellom Start og KLIKK NÅ – og tiden gjøres tilfeldig ved å bruke et tilfeldig tall fra rand-funksjonen. Resten av forklaringen på koden i dialogens vindusprosedyre er gitt som kommentarer i koden nedenfor.

NB! For å kunne bruke standardfunksjonen 'clock()' til å måle tiden som vi har gjort her, må det header-filen 'time.h' inkluderes:

```
#include <time.h>


// Mesage handler for tester dialog.
LRESULT CALLBACK TestDlgProc(HWND hDlg, UINT message,
                             WPARAM wParam, LPARAM lParam)
{
   switch( message )
   {
      // Hendler for knapp og Edit-kontroll, de må være static
      // fordi de settes ved mottak av WM_INITDIALOG og skal
      // brukes ved seinere kall til funksjonen.
      static HWND hKlikk;
      static HWND hTekst;

      int RandomTid;          // variabel ventetid
      static long T1;         // start på reaksjonsperiode
      long T2;                // slutt på reaksjonsperiode
      TCHAR szTekst[100];     // tekst i Edit-kontroll

      case WM_INITDIALOG:     // sendes ved åpning av dialog
         // GetDlgItem henter vindushendelen til en kontroll
         hTekst = ::GetDlgItem( hDlg, IDC_TEKST );
         ::SetWindowText( hTekst,
                          _T("Klikk på Start når du er klar.") );
         hKlikk = ::GetDlgItem( hDlg, IDC_KLIKK );
         return TRUE;

      case WM_COMMAND:
         switch(LOWORD(wParam))
         {
         case IDC_START:
            ::SetWindowText( hTekst, _T("Pass på ...") );
```

```
            RandomTid = 1000 + rand() / 10;
            ::SetTimer( hDlg, 1, RandomTid, NULL );
            return TRUE;

         case IDC_KLIKK:
            T2 = clock();      // clock() gir løpende tid i
                               // millisekunder - krever
                               // #include "time.h"
            wsprintf( szTekst, _T("Reaksjonstid: %d ms."),
                                                    T2-T1 );
            ::SetWindowText( hTekst, szTekst );
            ::ShowWindow( hKlikk, SW_HIDE );
            return TRUE;

         case IDOK:
         case IDCANCEL:
            EndDialog(hDlg, LOWORD(wParam));
            return TRUE;
         }
         break;

      case WM_TIMER:
         // gjør knappen synlig
         ::ShowWindow( hKlikk, SW_SHOW );

         T1 = clock();
         ::KillTimer( hDlg, 1 );
         return TRUE;
   }
    return FALSE;
}
```

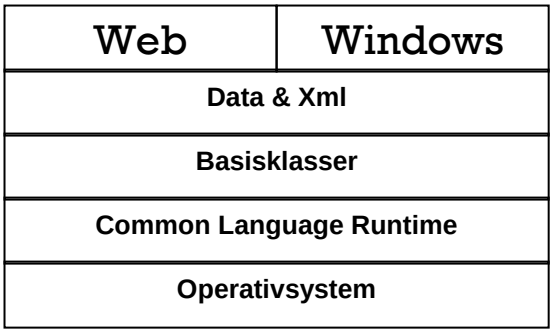
Bygg prosjektet og prøvekjør.

Hva er .NET Framework

Den teknologien som nå har fått navnet **.NET Framework** er ikke lett å karakterisere eller beskrive. Den er ikke et nytt operativsystem – den kan integreres i flere av de eksisterende Windows-variantene som finnes idag, og det er flere prosjekter igang for å implementere den på andre plattformer. Men den vil ha større betydning for hvordan man bygger og bruker programvare enn mange av oppgraderingene av operativ-systemene som vi har sett de siste årene.

Utgangspunktet for .NET går tilbake til ca 1998 da arbeidet med "neste versjon av COM" tok til. Erfaringene med COM hadde allerede da avdekke store problemer med denne teknologien i distribuerte systemer og med manglende skaler-barhet, nøkkelelementer for utfordringene fra Internet. Selv om vi har fått COM+ som til en viss grad representerer en fornyelse og videreføring av COM, ble det en gang for 4-5 år siden tatt en beslutning om å utvikle en fullstendig ny plattform for Microsofts programvareteknologi. Et stort prosjekt som nå begynner å levere.

.NET Framework er en ny plattform for komponent-basert programvare. Kjernen i den nye plattformen er det vi kan kalle "en ny virtuell maskin" plassert mellom operativsystemet og brukerprogrammene. Microsoft betegner sin implementasjon av denne virtuelle maskinen som "The Common Language Runtime", forkortet CLR. Den er en implementasjon av en spesifikasjon (CLI) som er overlatt til den internasjonale standardiseringsorganisasjonen ECMA og gjort åpent tilgjeng-elig for alle. Flere prosjekter for andre implementasjoner er underveis.



.NET Framework struktur

I tillegg til fokusen på distribuerte systemer i Internett, representerer innføringen av .NET Framework et forsøk på å forbedre programvareutvikling og –leveranse på flere andre områder som:

- sikkerhet
- færre feil
- enklere distribusjon og administrasjon av programmoduler.

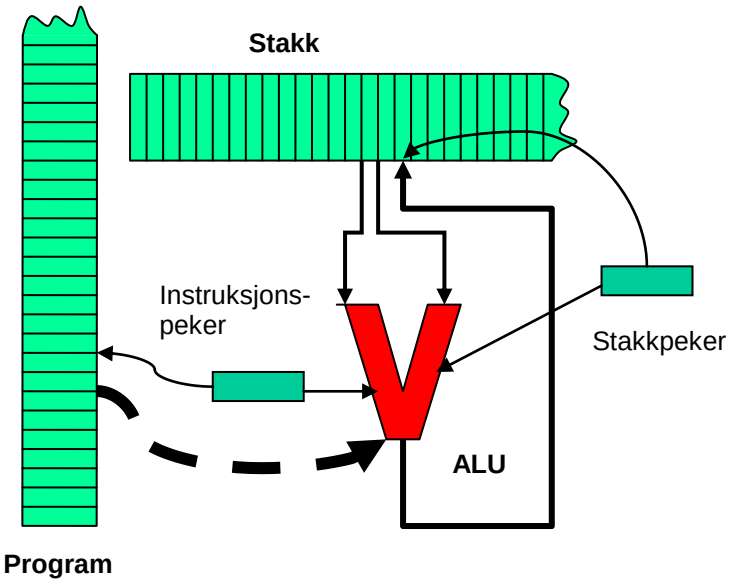
Common Language Runtime (CLR)

Common Language Runtime er betegnet som en virtuell maskin. Det er ikke et helt dekkende uttrykk. Det omfatter flere funksjoner og mange oppgaver.

Virtuell maskin

En virtuell maskin er en maskinarkitektur som kompilatorer arbeider mot når de konverterer fra høynivå kildekode til "maskinkode".

CLR kan beskrives fra flere perspektiv. Vi velger her å starte med Intermediate Language (IL) – et maskinspråk / instruksjonssett for en abstrakt, tenkt maskinarkitektur med stakk, hukommelse og en aritmetisk-logisk enhet (ALU).



GROV skisse av stakkbasert maskinarkitektur

Denne maskinarkitekturen definerer sammen med maskinspråket IL en generisk/generell datamaskinplattform som utgjør "kjernen" av CLR.

Dermed oppnår man:

- muligheter for plattformuavhengighet
- muligheter for språklig "interoperability" (at moduler fra ulike høynivåspråk kan koordineres og bindes sammen til en integrert kjørbar fil).

Som for enhver annen prosessor med et maskinspråk, så er det utviklet en assembler og en disassembler for IL-maskinen. En assembler er et program som leser "assembly kildekode" og produserer en eksekverbar programfil – en fil som kan lastes av maskinens "loader" og iverksettes. En disassembler kan lese en eksekverbar fil (maskinkode) og produsere assembly kildekode – litt mer forståelig enn maskinkode.

La oss skrive litt IL.

```
LDSTR "Hei"
CALL [mscorlib]System.Console.WriteLine(...)
RET
```

Alle kompilatorer for .NET Framework (C#, Visual Basic, Eiffel, …) vil produsere IL-kode. Det vil si at de produserer kode som ikke er låst til noen spesiell fysisk prosessor, men som er laget for en "virtuell maskin". Dette gir .NET en mulighet for "plattformuavhengighet" på samme måte som Java.

I tillegg til IL-kode produserer kompilatorene såkalte "meta-data" som også lagres i de eksekverbare filene. Metadata representerer en komplett beskrivelse av de datatypene som brukes i programmet, variabler, klasser, signaturer for medlemsfunksjoner, samt andre informasjoner som runtime-systemet kan dra nytte av for å kunne kjøre koden på en mye mer kontrollert og sikker måte enn tradisjonelt.

"Metadata" er ikke så nytt som det kanskje kan høres ut – mange av filtypene vi har operert med i lang tid inneholder metadata (data om data) – både vanlige EXE-filer og DLL-filer inneholder atskillig mer enn bare maskinkode.

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
"c:\winnt\>Depends Notepad.exe" → …
```

ILDASM (IL disassembler) viser både IL assemblykode og metadata.

--- eksempel
!!

”Loader” og ”runtime system” (kjøresystem)

Loader-funksjonen er tett integrert med operativsystemet og består i å laste opp programkode fra en eller flere filer, fylle ut "relocation adresses" og starte eksekvering.

Kjøresystemet er i hovedsak et sett av bibliotekkode som følger med operativsystemet og visse programutviklingsverktøy.

Programfilene som produseres med for .NET Framework får navn med filtypene .exe og .dll på samme måte som program-filer med tradisjonell maskinkode. De inneholder imidlertid IL-kode og metadata, og ikke maskinkode for den virkelige prosessoren.

Loaderen i CLR leser IL-koden og metadata inn i minnet. Først når en funksjon blir kalt blir funksjonens IL-kode omformet til ”virkelig maskinkode” eller ”native machine code”. Denne omformingen kalles ”Just in time”-kompilering.

Kompilering Just-In-Time

Programfilene fra .NET-kompilatorer inneholder IL-kode og metadata. Det er imidlertid bare maskinkode som kan eksekveres av en reell, fysisk prosessor, så IL-koden må før eller siden konverteres til maskinkode for den aktuelle prosessoren. CLR inneholder derfor en såkalt "JIT-kompilator" og konverteringen fra IL til maskinkode skjer ikke før det er helt nødvendig – nemlig når en funksjon kalles. Når en funksjon imidlertid er konvertert til maskinkode, blir maskinkoden tatt vare på, slik at neste kall til funksjonen kan gå rett inn i den virkelige maskinkoden.

Som en del av JIT-kompileringen må koden passere en verifikasjonsprosess (hvis ikke systemet er konfigurert annerledes). Verifikasjonen analyserer IL-koden for å sikre at den er ’type safe’ – det vil blant annet si at den garantert ikke aksesserer andre minneområder enn den er autorisert for. Systemet med slik kontroll av koden gjør det mulig å sikre seg mot en rekke former for vanlig feil, samt at det blir mulig å knytte ulike sikkerhetsbetingelser til eksekveringen av program, objekter og funksjoner.

Nå kan det også finnes korrekt og gyldig IL-kode som ikke passerer verifikasjonsprosessen som ’type-safe’. Dette skyldes begrensninger i hva som går an å bevise av korrekthet. Det finnes også programmeringsspråk som ”by design” produserer ”non-verifiable” eller ”unsafe” kode. Hvorvidt en vil tillate .NET Framework å kjøre programmer med ”unsafe” kode, blir da et valg som gjøres av systemadministrator for vedkommende maskin.

Kode som oppfyller kravene til å bli betraktet som ”type safe” og produseres på denne måten kalles "managed code".

Automatisk "Garbage Collection"

En av de mer betydningsfulle designvalgene som ble gjort under utviklingen av .NET Framework, var å basere seg på kjøresystemets Garbage Collection istedefor tradisjonell minnehåndtering slik vi kjenner den fra C/C++.

Kjøresystemet håndterer objektene og deres plassering i minnet og sørger automatisk for at minnet blir frigjort når objektene ikke lenger er i bruk. Objekter som blir kontrollert og administrert på denne måten, kalles ”managed data”. Hvis et program inneholder ’managed’ kode, så kan det også gjøre bruk av ’managed’ data. Men det kan også benyttes ’unmanaged’ data eller en blanding av de to typene data.

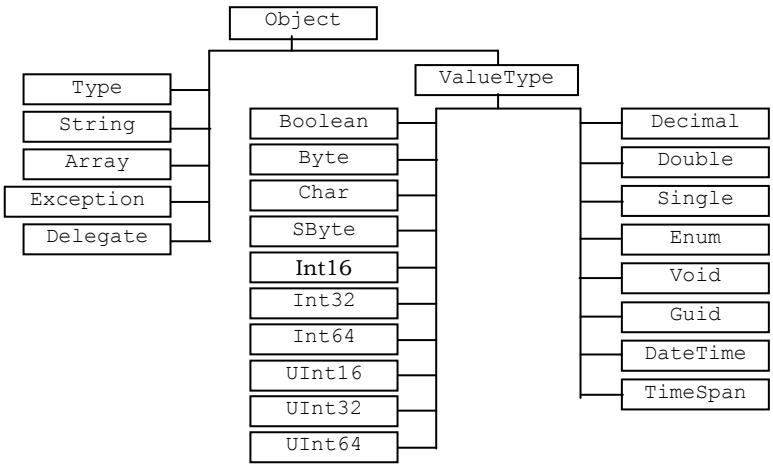
Common Type System - CTS

CTS spesifiserer hvordan datatypene deklareres, brukes og kontrolleres av kjøresystemet (CLR). Typesystemet omfatter følgende:

- et felles rammeverk for "typesafe" integrasjon av kode fra ulike programmeringsspråk
- en objektorientert modell som ulike programmeringsspråk kan bygge på
- definisjon av regler som de ulike språkene må etterkomme dersom den produserte koden skal kunne integreres med kode fra andre språk

Klassifikasjon av datatyper

CTS støtter to hovedkategorier av datatyper, som igjen er inndelt i en rekke underkategorier.



Figuren over viser hvordan alle predefinerte datatyper arver fra typen Object. Til venstre i figuren finner vi referansetypene som arver direkte fra Object, mens verditypene til høyre alle arver direkte fra typen ValueType som så arver fra Object.

Verdityper/Value types

Variabel-instanser av verditypene får innholdet sitt lagret direkte på stakken som den verdien de representerer – på samme måte som tradisjonelle lokale variabler. Dette kan være predefinerte typer fra CLR eller brukerdefinerte typer.

Referansetyper/Reference types

En instans av referansetypen har alltid en lagringsplass på heap'en for selve innholdet som variabelen representerer. Variabelen selv inneholder kun referansen (adressen) til lagringsplassen på heap'en.

Vi kommer tilbake til typesystemet i forbindelse med C#.

Klassebibliotekene i .NET Framework

Programbibliotekene i .NET Framework omfatter et rikt utvalg av klasser, interface og verdityper som gir tilgang til omfattende og avanserte systemfunksjoner. Bibliotekmodulene er bygget i henhold til Common Language Specification, så de kan benyttes av ulike språk og kompilatorer.

Bibliotekene er organisert i hierarkiske "Namespaces"/ navnerom for å unngå navnekollisjoner. En navnekollisjon inntreffer dersom det er brukt samme symbolnavn i ulike kodemoduler, og disse skal lenkes sammen i ett program. Et navnerom er en navngitt blokk av kode. Ved at ulike moduler bygges innenfor ulike navnerom vil man unngå slike kollisjoner selv om det er benyttet identiske symbolnavn. Ved lenkingen vil nemlig alle referanser være såkalte 'fullt kvalifiserte navn' – dvs. Navnerom.Symbolnavn.

Navnerommet System

Navnerommet System danner roten i den hierarkiske strukturen som klassebibliotekene er organiserte i. Der ligger klassene som representerer alle de grunnleggende datatypene samt andre grunnleggende klasser og datastrukturer.

På nivåene under System finnes en rekke mer og mer spesialiserte navnerom med flere biblioteksmoduler. Alt i alt skal det være 3-4000 klasser i bibliotekene. I oversikten nedenfor har vi listet opp endel av navnerommen under System sammen med en kort beskrivelse.

Category	Namespace	Functionality
Component model	System.ComponentModel	Implementation of components, including licensing and design-time adaptation.
Configuration	System.Configuration	Retrieval of application configuration data.
Data	System.Data	Access and management of data and data sources.
	System.Xml	Standards-based support for

		processing XML.
	System.Xml.Serialization	Bidirectional object–to-XML mapping.
Framework services	System.Diagnostics	Application instrumentation and diagnostics.
	System.DirectoryServices	Access to the Active Directory of an Active Directory service provider, such as Internet Information Services (IIS).
	System.Management	Services and application management tools that work with the Web-Based Enterprise Management (WBEM) standards.
	System.Timers	Event raising on an interval or more complex schedule.
Globalization and localization	System.Globalization	Support for internationalization and globalization of code and resources.