

## 2. Noțiuni de Win32 API ... pentru curajoși

Acronimul API este o abreviere a *Application Programming Interface*. Așadar Windows API (sau Win32 API) este un set de funcții oferite de sistemul de operare Windows pentru manipularea resurselor calculatorului. Orice sistem de operare oferă (sau exportă) un set de astfel de funcții, pentru a fi utilizate de programatori în dezvoltarea de aplicații specifice aceluși sistem de operare. Denumirea de *Win32 API* mai este folosită uneori pentru a marca diferența dintre sistemele de operare Windows pe 16 biți (Windows 3.X) și sistemele de operare Windows pe 32 de biți (Windows 9X, Windows NT, Windows 2000, Windows XP).

Funcțiile din interfața Win32 API sunt implementate în următoarele trei biblioteci: ***user32.dll***, ***kernel32.dll*** și ***gdi32.dll***. Ele sunt exportate și se pot folosi de către orice program, cu condiția includerii fișierului header corespunzător (care este ***Windows.h***) și a editării legăturilor programului cu fișierul de exporturi corespunzător (***user32.lib***, ***kernel32.lib*** sau/și ***gdi32.lib***). Nu vă speriați, aceste trei biblioteci sunt trecute în mod automat în lista de module a editorului de legături, în orice proiect generat de mediul Visual C++.

Descrierea interfeței Win32 API poate fi găsită în help-ul on-line la indexul *Platform SDK*. Funcțiile din interfața Win32 API sunt împărțite în mai multe secțiuni. Dintre acestea, cele mai importante sunt *User Interface Services* și *Windows Base Services*.

### 2.1. Câteva deosebiri între programele DOS și programele Windows

În mod evident, un program Windows și un program DOS sunt foarte ușor de deosebit în timpul execuției. Un program Windows va genera interfața de intrare-ieșire într-o fereastră, prin intermediul unor controale adecvate, pe când un program DOS va genera o interfață intrare-ieșire standard. De fapt, cu toate că fundamental programarea în C++ sub Windows și sub DOS este aceeași, diferențele dintre programele implementate sub cele două sisteme de operare sunt mult mai multe, în cele ce urmează fiind enumerate câteva dintre ele:

- în DOS, pointerii erau mărimi reprezentate pe 16 biți. Cu alte cuvinte, prin intermediul unui pointer se putea reprezenta un spațiu de maximum 64 Kb de memorie. Aceasta provine din faptul că sistemul de operare DOS împarte memoria calculatorului în segmente de câte 64 de Ko, un pointer obișnuit (***NEAR***) putând genera adrese doar în interiorul segmentului din care face parte. Pentru a accesa o adresă din alt segment, este necesară generarea unui pointer ***FAR*** (pentru aceasta există funcția ***MK\_FP()***). În Windows nu mai există segmentarea memoriei. Adresele sunt generate implicit pe 32 de biți, deci, în consecință și pointerii vor fi reprezentați pe 32 de biți.
- DOS este un sistem de operare monotasking, adică la un moment dat poate rula un singur program. Windows este un sistem de operare multitasking, adică mai multe programe se pot afla în execuție simultan. Deci, în memoria calculatorului sunt încărcate simultan mai multe programe, fiecare având bine definit spațiul

propriu de adrese. Generarea unui pointer care adresează memoria în afara spațiului alocat programului duce la consecințe dezastruoase.

- Windows este un sistem de operare bazat pe interfețe grafice. Utilizarea funcțiilor de intrare-ieșire `scanf()`, `printf()`, `cin` și `cout` nu va genera erori de compilare, dar nu va avea nici un efect în cadrul programului executabil.
- în DOS, intrările și ieșirile din programe sunt sincrone cu desfășurarea lor, adică se pot introduce date doar atunci când programul execută instrucțiuni de citire și respectiv, se pot afișa date, doar atunci când programul execută instrucțiuni de afișare. În Windows, pe lângă intrările-ieșirile sincrone, un program poate prelua și respectiv afișa date și asincron, în momentul producerii unui *eveniment*. Prin eveniment se înțelege orice modificare în starea programului (acțiuni ale utilizatorului, comenzi ale sistemului de operare, scurgerea unui interval de timp, modificarea valorii unor mărimi fanion, etc).

## 2.2 Tipuri de date noi folosite de funcțiile Win32 API

Datorită diferențelor dintre programele DOS și Windows, Win32 API va utiliza o serie de tipuri de date noi. Câteva din acestea sunt:

- **HANDLE** – este un tip generic (*identificator*), utilizat pentru manipularea obiectelor folosite în program (fișiere, ferestre, etc.). Pentru a manipula un astfel de obiect, este necesară întâi obținerea unui asemenea **HANDLE**, în urma apelării unei funcții care îl returnează. Toate operațiile ulterioare cu obiectul respectiv se vor face prin intermediul mărimii **HANDLE** și nu prin intermediul numelui obiectului respectiv;
- **HWND** - este definit ca `typedef HANDLE HWND;` și este folosit pentru manipularea ferestrelor;
- **DWORD** – (**Double Word**) este un întreg fără semn pe 32 de biți. Uzual, un **DWORD** este compus din două mărimi **WORD** și marea majoritate a compilatoarelor permit extragerea celor două componente **WORD** prin funcții de tipul `high()` și `low()`;
- **LPVOID** – (**Long Pointer Void**) este definit ca `typedef void* LPVOID`, fiind deci un pointer `void` reprezentat pe 32 de biți;
- **LPCTSTR** – (**Long Pointer Constat To String**) – reprezintă un pointer pe 32 de biți spre un șir de caractere constant. Este utilizat de obicei atunci când șirul este utilizat ca parametru al unei funcții și funcția nu îl modifică;
- **LPCTSTR** – (**Long Pointer Constat To String**) – reprezintă un pointer pe 32 de biți spre un șir de caractere constant *Unicode*; Unicode este un cod de caractere pe 16 biți, capabil să reprezinte caracterele tuturor limbilor. Este utilizat de platformele Windows NT (2000, XP). Windows 95, 98 și Milenium nu îl utilizează. Pentru a defini un pointer similar, dar spre un șir ASCII, vom declara tipul **LPCSTR**;
- **LPCSTR** – (**Long Pointer Constat String**) – reprezintă un pointer pe 32 de biți spre un șir de caractere constant ASCII; se utilizează în aceleași situații ca și tipul anterior;
- **LPTSTR** – (**Long Pointer To String**) – reprezintă un pointer pe 32 de biți spre un șir de caractere în format *Unicode*;
- **LPSTR** – (**Long Pointer String**) – reprezintă un pointer pe 32 de biți spre un șir de caractere în format ASCII;
- **WPARAM** și **LPARAM** – cuvinte cu lungimea de 32 de biți, utilizate în general pentru a transmite parametri asociați unui mesaj Windows;
- **LRESULT** – o valoare pe 32 de biți, returnată de o funcție;

## 2.3 Ferestre ... lucrăm sub Windows

O fereastră este elementul fundamental, pe baza căruia se construiește orice interfață grafică utilizator. Interfața sistemelor de operare Windows este bazată pe ferestre. Majoritatea aplicațiilor Windows posedă o fereastră principală, în care rulează. O fereastră alocă unei aplicații un spațiu de formă dreptunghiulară pe ecran, ca în fig. 2.1.

O fereastră tipică este construită din 7 elemente de bază, care apoi pot fi fragmentate de programele pe care le vom construi, astfel încât unele din ele pot să lipsească, sau pot să apară de mai multe ori. Aceste elemente sunt:

- **cadrul ferestra** (*Window Frame, Main Frame*) – este în general zona activă care mărginește fereastra și permite redimensionarea ei. Reprezintă containerul principal pentru toate celelalte componente ale ferestrei;
- **bara de titlu** (*Title Bar*) - afișează titlul aplicației și eventual denumirea documentului deschis pentru prelucrare. Este o zonă activă, care permite mutarea ferestrei pe ecran. Conține de asemenea butoanele de minimizare, maximizare și închidere a ferestrei;
- **butoanele de minimizare, maximizare și închidere** - apar de obicei în bara de titlu;
- **bara de meniu** (*Menu Bar*) - conținut tot de bara de titlu (în partea stângă a acesteia). Conține unul sau mai multe meniuri pe baza cărora se implementează diferite funcții ale programului;
- **bara de defilare** (*Scroll Bar*) – permite utilizatorului defilarea sus-jos, respectiv dreapta-stânga în cadrul ferestrei;
- **bara de stare** (*Status Bar*) – oferă utilizatorului informații specifice din program;
- **zona client a ferestrei** (*Client Area*) - este zona pusă la dispoziție pentru aplicația ce rulează în fereastra respectivă;

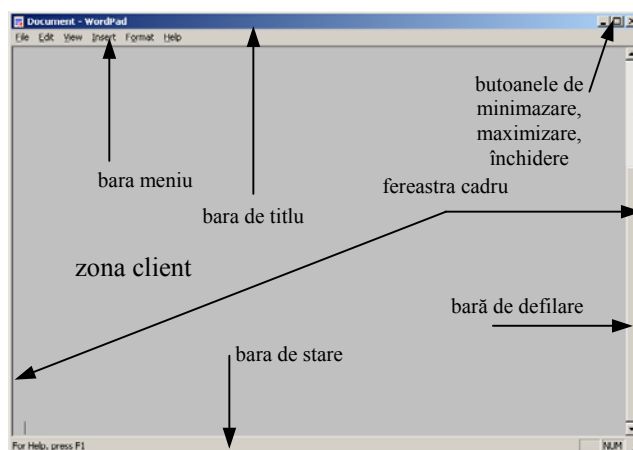


Figura 2.1 O fereastră tipică...

Ferestrele posedă două dimensiuni: X, pe orizontală și Y, pe verticală. Cu toate acestea, mulțimea ferestrelor afișate pe ecran la un moment dat trebuie privită ca fiind în spațiu. Toate ferestrele au o a treia dimensiune comună, Z, și sunt caracterizate de o anumită ordine în care sunt afișate pe axa Z. De exemplu, atunci când apăsăm click stânga în interiorul unei ferestre care nu era activă și era parțial ascunsă, aceasta este activată (adică i se desenează bara de titlu cu o altă culoare) și se schimbă ordinea Z (z

*order*) a ferestrelor, de așa manieră încât fereastra activată devine ultima fereastră din ordinea Z (fereastra din față).

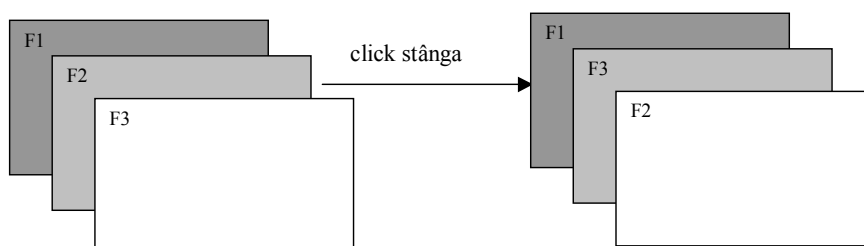


Figura 2.2. Manipularea ferestrelor pe axa Z

Figura 2.2 prezintă o asemenea situație. De reținut faptul că la un moment dat, o singură fereastră poate fi activă. O fereastră care este activă, pe lângă desenarea ei ca ultima fereastră în ordinea Z, primește *input focus*-ul, adică poate recepționa evenimentele produse la tastatură.

Putem distinge două tipuri de ferestre: *top-level* (ferestrele principale în care rulează o aplicație) și *child* (ferestre copil, de exemplu butoanele și alte elemente de control dintr-o casetă de dialog sunt ferestre copil ale casetei de dialog).

O fereastră este unic identificată în sistem printr-un identificator (o variabilă de tip `HWND`). Acest identificator se obține în urma apelării funcției API `CreateWindow()`.

## 2.4 Mesaje Windows. Tratarea evenimentelor asincrone

Orice eveniment asincron (scurgerea unui interval de timp, apăsarea unei taste, mutarea mouse-lui, etc) este recepționat de către sistemul de operare Windows și este transformat într-un *mesaj*. Acest mesaj este transmis apoi fie către sistemul de operare, fie către programul căruia îi este destinat, pentru a se executa o acțiune în concordanță cu evenimentul produs. O schemă simplificată a acestui mecanism este prezentată în fig. 2.3.

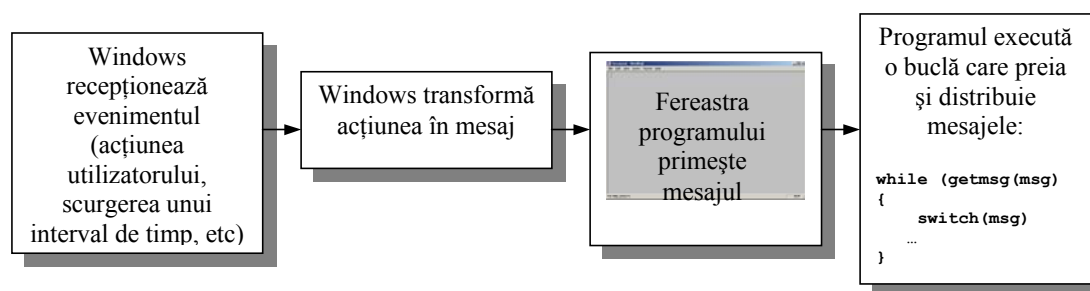


Figura 2.3 Așa tratează Windows mesajele

Toate mesajele Windows au prefixul `WM_`, urmat de o denumire care pune în evidență evenimentul care l-a produs. Exemple de astfel de mesaje sunt: `WM_KEYDOWN` – mesaj produs de apăsarea unei taste; `WM_MOUSEMOVE` – mesaj produs de mișcarea mouse-lui; `WM_LEFTBUTTONDOWN` – mesaj produs de apăsarea butonului stâng al mouse-lui; `WM_CLOSE` – mesaj produs de apăsarea butonului de închidere a ferestrei, etc. Unele din aceste mesaje, trebuie să ofere și informații suplimentare, cum ar fi: `WM_KEYPRESSED` trebuie să transmită codul ASCII și codul de scanare al tastei apăsate; `WM_MOUSEMOVE` coordonatele X și Y al cursorului mouse-lui; `WM_LEFTBUTTONDOWN` coordonatele X și Y ale cursorului mouse-lui în care a fost apăsat butonul stâng, etc.

Alte mesaje, cum este de exemplu `WM_CLOSE` nu trebuie să transmită nici o informație suplimentară.

În momentul primirii unui mesaj Windows, programul căruia îi este destinat trebuie să identifice mesajul și să execute secvența de instrucțiuni corespunzătoare. Deci, o funcție foarte importantă într-un program este aceea care acceptă și prelucrează mesajele transmise de sistemul de operare. Pentru ferestrele “clasice”, această funcție este o funcție specială, numită *procedura fereastră*:

```
LRESULT CALLBACK WndProc(HWND hwnd,
                          UINT uMsg,
                          WPARAM wParam,
                          LPARAM lParam);
```

Ce observăm referitor la această funcție? În primul rând, ea returnează o valoare `LRESULT`. Această valoare reprezintă rezultatul procesării mesajului și depinde de mesajul trimis. Ca o noutate, această funcție este declarată `CALLBACK`. O funcție `CALLBACK` este o funcție care recepționează mesaje de la sistemul de operare. Cum `WndProc()` are tocmai rolul de a prelua și prelucra mesajele, este normal ca ea să fie declarată `CALLBACK`.

Funcția primește următoarele argumente:

- `HWND hwnd` – identificatorul ferestrei (sau programului, căci să nu uităm, orice program are atașată cel puțin o fereastră) pentru care se preiau mesajele;
- `UINT uMsg` – mesajul transmis de sistemul de operare;
- `WPARAM wParam, LPARAM lParam` – parametrii mesajului;

În funcție de mesajul primit, funcția va executa prelucrarea corespunzătoare, uzual prin intermediul unui selector `switch`.

## 2.5 Înregistrarea clasei de ferestre

Fereastra pe care dorim să o asociem programului, în general, nu este unică. Ea este creată pe baza unor șabloane, la fel cu mai multe ferestre care pot avea aceleași caracteristici (de exemplu dimensiune, stare inițială, dacă se pot redimensiona sau nu, etc.). Se spune că un grup de astfel de ferestre formează o *clasă de ferestre*. O clasă de ferestre poate fi considerată un șablon pe baza căruia se crează apoi ferestre. Pentru a putea folosi o clasă de ferestre, este necesară înregistrarea ei, cu funcția API `RegisterClass()`. La înregistrarea unei clase de ferestre, se specifică, printre alte caracteristici și procedura fereastră care se va folosi de către ferestrele din clasa respectivă. Funcția `RegisterClass()` este declarată ca și:

```
ATOM RegisterClass(CONST WNDCLASS *lpWndClass);
```

Funcția returnează o valoare de tip `ATOM`. Tipul `ATOM` este un `WORD` care reprezintă o referință la un șir de caractere, indiferent dacă șirul conține majuscule sau minuscule. Astfel, spre exemplu, șirul “visual c” va fi echivalent cu șirul “VISUAL C”. Funcția primește de asemenea ca și parametru un pointer la o structură de tip `WNDCLASS`. Această structură definește de fapt felul în care va arăta o fereastră din clasa respectivă:

```
typedef struct _WNDCLASS {
    UINT style;
    WNDPROC lpfnWndProc;
    int cbClsExtra;
    int cbWndExtra;
    HANDLE hInstance;
    HICON hIcon;
    HCURSOR hCursor;
    HBRUSH hbrBackground;
    LPCTSTR lpszMenuName;
    LPCTSTR lpszClassName;
} WNDCLASS;
```

Câmpurile structurii au următoarele semnificații:

- `UINT style` – reprezintă un stil sau o combinație de stiluri, obținută prin intermediul operatorului `OR`. Câteva din stilurile posibile sunt prezentate în tabelul 2.1:

**Tabelul 2.1**

| Stil                            | Semnificație  |
|---------------------------------|---|
| <code>CS_BYTEALIGNWINDOW</code> | aliniază fereastra pe orizontală în poziția dată de <code>BYTE</code>             |
| <code>CS_DBLCLKS</code>         | înștiințează fereastra când utilizatorul execută un dublu click cu mouse-le       |
| <code>CS_HREDRAW</code>         | redesenează întreaga fereastră când utilizatorul ajustează dimensiunea orizontală |
| <code>CS_VREDRAW</code>         | redesenează întreaga fereastră când utilizatorul ajustează dimensiunea verticală  |
| <code>CS_NOCLOSE</code>         | dezactivează comanda și butonul <code>Close</code>                                |

- `WNDPROC lpfnWndProc` – numele funcției `callback` care prelucrează mesajele furnizate de Windows;
- `int cbClsExtra` – un număr de octeți suplimentari ce trebuie alocați la sfârșitul structurii pentru stocarea de informații;
- `int cbWndExtra` – un număr de octeți suplimentari ce trebuie alocați la crearea fiecărei instanțe a clasei pentru stocarea de informații;
- `HANDLE hInstance` – identificator pentru instanța de care aparține clasa fereastră;
- `HICON hIcon` – un identificator pentru pictograma folosită de fereastră;
- `HCURSOR hCursor` – un identificator pentru cursorul folosit de fereastră;
- `HBRUSH hbrBackground` – un identificator pentru pensula utilizată pentru colorarea zonei client a ferestrei;
- `LPCTSTR lpszMenuName` – un pointer spre un șir de caractere terminat cu “\0” care reprezintă numele meniului asociat clasei. Dacă este `NULL`, se încarcă meniul implicit al clasei fereastră;
- `LPCTSTR lpszClassName` – un pointer spre un șir de caractere (terminat cu “\0”) care reprezintă numele clasei. Acest nume va fi folosit ulterior pentru crearea unei instanțe a clasei, adică a unei ferestre de acest tip.

## 2.6 Crearea unei ferestre

Nu vom putea executa un program sub Windows dacă nu vom crea mai întâi o fereastră. Crearea unei ferestre se face cu ajutorul funcției

```

HWND CreateWindow(
    LPCTSTR lpClassName,
    LPCTSTR lpWindowName,
    DWORD dwStyle,
    int x,
    int y,
    int nWidth,
    int nHeight,
    HWND hWndParent,
    HMENU hMenu,
    HANDLE hInstance,
    LPVOID lpParam
);

```

Observăm că, crearea ferestrei este reușită, funcția returnează un identificator la fereastra respectivă. Identificator va fi utilizat apoi de către orice altă operație asupra ferestrei pentru a o identifica. De altfel, este identificatorul pe care-l va primi și procedura fereastră asociată ferestrei nou create. Pentru crearea unei noi ferestre, va trebui să specificăm următorii parametri:

- `LPCTSTR lpClassName` – un pointer spre un șir de caractere, reprezentând un nume valid de fereastră. Uzual este fie șirul `lpClassName` definit în faza de înregistrare a clasei fereastră, fie un nume predefinit de fereastră (`BUTTON`, `LISTBOX`, `EDIT`, `COMBOBOX`, `STATIC`, etc);
- `LPCTSTR lpWindowName` – un pointer spre un șir de caractere care conține numele ferestrei. Este în general o etichetă asociată ferestrei și în funcție de stilul acesteia poate fi plasată în diferite locuri;
- `DWORD dwStyle` – o valoare `DWORD` reprezentând stilul ferestrei. Mai multe stiluri pot fi combinate prin intermediul operatorului `OR`. Câteva din stilurile posibile sunt prezentate în tabelul 2.2:

Tabelul 2.2

| Stil                             | Semnificație   |
|----------------------------------|--|
| <code>WS_BORDER</code>           | crează o fereastră cu o linie subțire ca margine   |
| <code>WS_CAPTION</code>          | crează o fereastră care are o bară de titlu  |
| <code>WS_DISABLED</code>         | crează o fereastră inițial dezactivată. O astfel de fereastră nu reacționează la nici un stimul de intrare din partea utilizatorului   |
| <code>WS_HSCROLL</code>          | crează o fereastră cu bară de defilare orizontală  |
| <code>WS_MAXIMIZE</code>         | crează o fereastră inițial maximizată  |
| <code>WS_MAXIMIZEBOX</code>      | crează o fereastră care are butonul de maximizare  |
| <code>WS_MINIMIZE</code>         | crează o fereastră inițial minimizată  |
| <code>WS_MINIMIZEBOX</code>      | crează o fereastră care are butonul de minimizare  |
| <code>WS_OVERLAPPED</code>       | crează o fereastră cu bară de titlu și bordură   |
| <code>WS_OVERLAPPEDWINDOW</code> | crează o fereastră cu stilurile implicite <code>WS_OVERLAPPED</code> , <code>WS_CAPTION</code> , <code>WS_SYSMENU</code> , <code>WS_THICKFRAME</code> , <code>WS_MINIMIZEBOX</code> și <code>WS_MAXIMIZEBOX</code> |
| <code>WS_VISIBLE</code>          | crează o fereastră inițial vizibilă  |
| <code>WS_VSCROLL</code>          | crează o fereastră cu bară de defilare verticală   |

- `int x` – poziția orizontală a colțului stânga sus a ferestrei;
- `int y` – poziția verticală a colțului stânga sus a ferestrei;
- `int nWidth` – lungimea pe orizontală a ferestrei;
- `int nHeight` – înălțimea pe verticală a ferestrei; **Observație: pentru acești 4 ultimi parametri, dacă valorile nu sunt importante se trece valoarea `CW_USERDEFAULT`;**

- `HWND hWndParent` – un identificator al ferestrei părinte. Dacă fereastra este primară, deci nu este lansată din altă fereastră, acest identificator este `NULL`;
- `HMENU hMenu` – un identificator al meniului ferestrei. Dacă clasa fereastră are un meniu implicit și fereastra îl folosește, se trece `NULL`;
- `HANDLE hInstance` – identificatorul instanței programului care a produs fereastra. Este necesar, deoarece în Windows pot fi în execuție simultan mai multe instanțe ale aceluiași program (spre exemplu, dacă lansați simultan 3 Internet Explorer);
- `LPCVOID lpParam` – un pointer spre un șir de parametri oferiți ferestrelor copil (numai pentru aplicații **Multiple Document Interface**);

## 2.7 Fereastra trebuie să fie vizibilă! Funcția `ShowWindow()`

Până în acest moment am declarat o clasă de ferestre și am creat o instanță a acesteia, deci o nouă fereastră. Fereastra nou creată există ca obiect în memorie, dar ea trebuie și afișată pe ecran. Afișarea se face cu ajutorul funcției `ShowWindow()`:

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

Funcția returnează `TRUE` în caz de succes și primește următoarele argumente:

- `HWND hWnd` – identificatorul ferestrei nou create, ce trebuie afișate;
- `int nCmdShow` – un parametru care specifică modul de afișare a ferestrei. Câteva din valorile posibile pentru acest parametru sunt prezentate în tabelul 2.3:

**Tabelul 2.3**

| Valoare                  | Semnificație  |
|--------------------------|---|
| <code>SW_HIDE</code>     | ascunde fereastra și activează o alta, dacă există              |
| <code>SW_MAXIMIZE</code> | maximizează fereastra specificată                               |
| <code>SW_MINIMIZE</code> | minimizează fereastra specificată                               |
| <code>SW_RESTORE</code>  | activează și afișează fereastra în starea în care a fost creată |
| <code>SW_SHOW</code>     | face fereastra vizibilă   |

## 2.8 Și acum, să prelucrăm mesajele ...

Evenimentele la care trebuie să răspundă un program, deci implicit mesajele recepționate de acesta pot apare într-un ritm mult mai rapid decât este programul capabil să prelucreze. Totuși, nici un mesaj nu se pierde, deoarece fiecare mesaj este stocat într-o **coadă de mesaje**. Programul va extrage apoi pe rând mesajele din coada de mesaje și le va prelucra în mod corespunzător, prin intermediul procedurii fereastră.

Spre exemplu, mecanismul de generare a mesajelor de către Windows la apăsarea tastelor și memorarea lor în coada de mesaje este prezentat în fig. 2.4. La introducerea textului **SALUT!**, evenimentele generate de apăsarea tastelor sunt plasate în coada de mesaje. De acolo sunt extrase de către bucla de mesaje asociată programului și este generat mesajul `WM_CHAR` care este transmis pentru prelucrare procedurii fereastră.



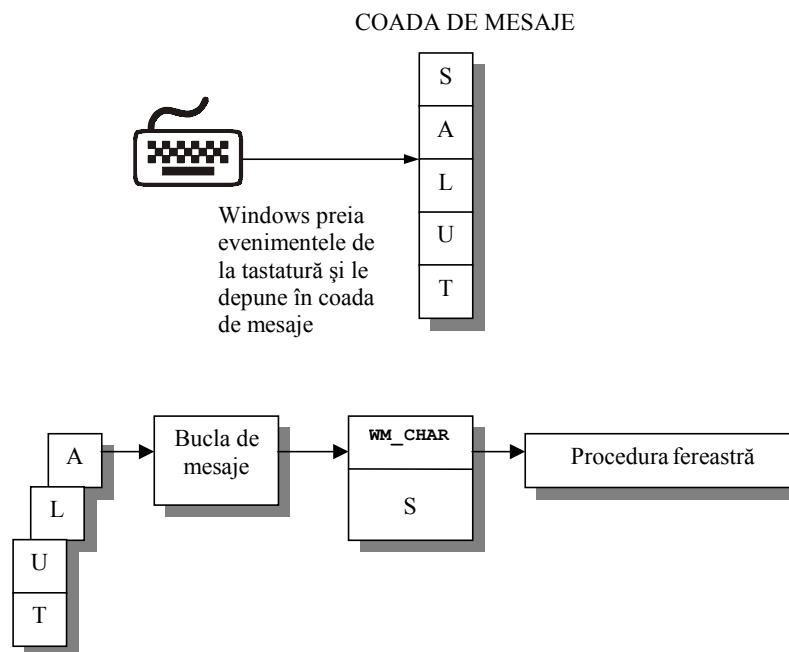


Figura 2.4. Așa se tratează mesajele

Pentru preluarea mesajelor din coada de mesaje, orice program Windows va trebui să conțină o **bucă de mesaje** care să preia și să proceseze mesajele. Această buclă de mesaje este un ciclu `while`, compus din 3 funcții, cu implementarea de mai jos:

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

Prima funcție utilizată, care stabilește condiție de ieșire din bucla de mesaje este

```
BOOL GetMessage(
    LPMSG lpMsg, HWND hWnd, UINT wParamFilterMin, UINT wParamFilterMax);
```

Funcția recepționează mesajele de la sistemul de operare și le stochează într-o structură `MSG`. Ea returnează `TRUE` atâta timp cât nu se primește mesajul `WM_QUIT` de terminare a programului. Cu alte cuvinte, orice program Windows va executa bucla de mesaje până la primirea din partea sistemului de operare a mesajului de terminare. Parametrii funcției au semnificațiile:

- `LPMSG lpMsg` – un pointer la structura `MSG`;
- `HWND hWnd` – un identificator pentru fereastra care primește mesajul. O valoare `NULL` forțează `GetMessage()` să rețină toate mesajele;
- `UINT wParamFilterMin` – valoarea minimă a mesajului primit (de obicei 0);
- `UINT wParamFilterMax` – valoarea maximă a mesajului primit (de obicei tot 0, pentru ca `GetMessage()` să accepte toate mesajele);

Structura `MSG` are declarația

```
typedef struct tagMSG {
    HWND    hwnd;
```

```

    UINT    message;
    WPARAM  wParam;
    LPARAM  lParam;
    DWORD   time;
    POINT    pt;
} MSG;

```

câmpurile având următoarele semnificații:

- `HWND hwnd` – identificatorul ferestrei pentru care procedura fereastră asociată primește mesajele;
- `UINT message` – valoarea mesajului;
- `WPARAM wParam, LPARAM lParam` – valorile parametrilor asociați mesajului;
- `DWORD time` – momentul trimiterii mesajului;
- `POINT pt` – poziția cursorului în coordonate ecran (referitor la colțul stânga sus al ecranului) în momentul în care a fost trimis mesajul.

Prima funcție utilizată în interiorul buclei de mesaje este `TranslateMessage()`. Ea este declarată ca:

```
BOOL TranslateMessage(CONST MSG *lpMsg);
```

cu parametrul

- `CONST MSG *lpMsg` – un pointer la structura `MSG` care stochează mesajul.

Această funcție are rolul de a prelua mesajele de la tastele virtuale și a le încapsula în mesajul `WM_CHAR`, pe care apoi îl pune în coada de mesaje. Acest mesaj este transmis de Windows ori de câte ori este apăsată o tastă care asociază un cod ASCII. De asemenea mesajul este generat după unul din mesajele `WM_KEYDOWN`, `WM_KEYUP`, `WM_SYSKEYDOWN`, or `WM_SYSKEYUP`. Mesajul încapsulează două valori:

- `chCharCode = (TCHAR) wParam` – codul ASCII al tastei apăsată;
- `lKeyData = lParam` – codul de scanare (poziția în matricea de taste) a tastei apăsată;

Funcția returnează `TRUE` ori de câte ori este apăsată o astfel de tastă și generează mesajul `WM_CHAR`. Dacă mesajul transmis de sistemul de operare nu se referă la o tastă virtuală, funcția `TranslateMessage()` va ignora mesajul. Câteva din codurile tastelor virtuale (în afară de tastele caracter obișnuite) sunt prezentate în tabelul 2.4:

**Tabelul 2.4**

| Definiție               | Tasta asociată  |
|-------------------------|-----------------|
| <code>VK_BACK</code>    | Backspace       |
| <code>VK_TAB</code>     | Tab             |
| <code>VK_LEFT</code>    | Săgeată stânga  |
| <code>VK_RIGHT</code>   | Săgeată dreapta |
| <code>VK_UP</code>      | Săgeată în sus  |
| <code>VK_DOWN</code>    | Săgeată în jos  |
| <code>VK_RETURN</code>  | Enter           |
| <code>VK_ESCAPE</code>  | Esc             |
| <code>VK_CONTROL</code> | Ctrl            |

Cea de a doua funcție utilizată în bucla de mesaje este `DispatchMessage()` declarată ca:

```
LONG DispatchMessage(CONST MSG *lpmsg);
```

având același parametru de intrare ca și funcția precedentă. Această funcție trimite mesajul procedurii fereastră pentru prelucrare.

## 2.9 Să scriem un program Windows simplu

Ca o concluzie la cele arătate anterior, rezultă că pentru scrierea unui program Windows simplu va trebui să parcurgem următorii pași:

- să declarăm (eventual într-un fișier header) funcția callback, în cazul nostru `WndProc()`;
- să declarăm și să definim clasa de ferestre utilizate ( în fișierul sursă);
- să creem o instanță a acestei clase, utilizând funcția `CreateWindow()`;
- să stabilim starea de vizibilitate a ferestrei cu funcția `ShowWindows()`;
- să implementăm bucla de mesaje;
- să implementăm procedura fereastră;

Pentru exemplificare, vom crea un nou proiect, numit *PrimaFereastră*. De data aceasta, proiectul nu va fi de **tip Win32 Console Application**, ci **Win 32 Application**. Și de această dată, vom alege la pasul 1 al wizardului opțiunea **An empty project**. Vom adăuga proiectului fișierul *PrimaFereastră.h*, cu conținutul:

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

Cu alte cuvinte, am declarat procedura fereastră. Urmează adăugarea la proiect și implementarea fișierului sursă, *PrimaFereastră.cpp*:

```
#include <Windows.h>
#include "PrimaFereastra.h"

int WINAPI WinMain( HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPSTR lpCmdLine,
    int nCmdShow
)
{
    HWND hwndMain;
    MSG msg;
    WNDCLASS wndCls;
    UNREFERENCED_PARAMETER(lpCmdLine);

    wndCls.style = 0;
    wndCls.lpfnWndProc = (WNDPROC) WndProc;
    wndCls.cbClsExtra = 0;
    wndCls.cbWndExtra = 0;
    wndCls.hInstance = hInstance;
    wndCls.hIcon = LoadIcon((HINSTANCE) NULL, IDI_APPLICATION);
    wndCls.hCursor = LoadCursor((HINSTANCE) NULL, IDC_ARROW);
```

```

wndCls.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
wndCls.lpszMenuName = NULL;
wndCls.lpszClassName = "MainWndClass";

if (!RegisterClass(&wndCls))
    return FALSE;

hwndMain = CreateWindow("MainWndClass", "PrimaFereastră",
    WS_OVERLAPPEDWINDOW | WS_HSCROLL | WS_VSCROLL, CW_USEDEFAULT,
    CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, (HWND) NULL,
    (HMENU) NULL, hInstance, (LPVOID) NULL);

if (!hwndMain)
    return FALSE;
ShowWindow(hwndMain, nCmdShow);
UpdateWindow(hwndMain);

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

LRESULT CALLBACK WndProc(HWND hwnd, UINT uMsg,
    WPARAM wParam, LPARAM lParam)
{
    switch(uMsg) {
    case WM_CLOSE:
        PostMessage(hwnd, WM_QUIT, 0L, 0L);
        return DefWindowProc(hwnd, uMsg, wParam, lParam);
    default:
        return DefWindowProc(hwnd, uMsg, wParam, lParam);
    }
}

```

Să comentăm puțin programul. În primul rând, funcția principală a programului nu mai este `main()`, ci `WinMain()`. Această funcție returnează întotdeauna la terminare o valoare, care este uzual valoarea câmpului `wParam` din structura `MSG`. `WinMain()` nu poate fi utilizat fără cele 4 argumente asociate.

Programul declară identificatorul ferestrei asociate programului, coada de mesaje asociată și clasa de ferestre utilizată. După care, construiește aspectul clasei de ferestre și înregistrează clasa respectivă. Dacă înregistrarea eșuează, programul se termină returnând codul de eroare `FALSE`.

După înregistrarea clasei, este creată fereastra programului apelând funcția `CreateWindow()`. Această funcție returnează în caz de creare reușită un identificator al ferestrei. În caz contrar, identificatorul este încărcat cu `NULL` și programul se termină. Pe baza identificatorului, este stabilită starea de vizibilitate a ferestrei cu funcția `ShowWindow()` și apoi este reactualizată zona client a ferestrei (fundal, conținut, etc) cu ajutorul funcției `UpdateWindow()`. În final, programul execută bucla de mesaje, iar la primirea mesajului `WM_QUIT` își încheie execuția.

Procedura fereastră tratează doar cazul mesajului `WM_CLOSE`, care termină programul. În cazul apariției acestui mesaj, se apelează funcția `PostMessage()` pentru transmiterea mesajului `WM_QUIT`. Funcția `PostMessage()` cu declarația

```
BOOL PostMessage(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam);
```

având semnificațiile cunoscute pentru parametri, plasează mesajul specificat în coada de mesaje asociată programului.

Să vedem cum putem interpreta și alte mesaje. Să modificăm programul, astfel încât la apăsarea tastelor descrise în tabelul 2.4, să ne fie afișat un mesaj care să specifice ce tastă a fost apăsată. Afișarea mesajului se face cu funcția

```
int MessageBox(HWND hWnd, LPCTSTR lpText,
               LPCTSTR lpCaption, UINT uType);
```

unde:

- HWND hWnd – identificatorul ferestrei în care se va afișa mesajul;
- LPCTSTR lpText – mesajul afișat;
- LPCTSTR lpCaption – eticheta afișată în bara de titlu;
- UINT uType – tipul ferestrei utilizate pentru afișare;

Ce va trebui să modificăm? Evident, procedura fereastră, astfel încât să trateze și mesajul WM\_KEYDOWN. Pentru acest mesaj, parametrul wParam conține codul tastei apăsată.

```
LRESULT CALLBACK WndProc( HWND hwnd, UINT uMsg,
                          WPARAM wParam, LPARAM lParam)
{
    switch(uMsg){
    case WM_CLOSE:
        PostMessage(hwnd, WM_QUIT, 0L, 0L);
        return DefWindowProc(hwnd, uMsg, wParam, lParam);
    case WM_KEYDOWN:
        LPCTSTR text;
        switch(wParam){
            case VK_BACK: text="Ati apasat tasta BackSpace"; break;
            case VK_TAB: text="Ati apasat tasta Tab"; break;
            case VK_LEFT: text="Ati apasat tasta SageataStinga";
                break;
            case VK_RIGHT: text="Ati apasat tasta SageataDreapta";
                break;
            case VK_UP: text="Ati apasat tasta SageataSus"; break;
            case VK_DOWN: text="Ati apasat tasta SageataJos"; break;
            case VK_RETURN: text="Ati apasat tasta Enter"; break;
            case VK_ESCAPE: text="Ati apasat tasta Escape"; break;
            case VK_CONTROL: text="Ati apasat tasta Control"; break;
            default: text="Ati apasat alta tasta";
                break;
        }
        MessageBox(hwnd, text, "Mesaj", MB_OK|MB_ICONEXCLAMATION);
        return DefWindowProc(hwnd, uMsg, wParam, lParam);
        break;
    default:
        return DefWindowProc(hwnd, uMsg, wParam, lParam);
    }
}
```

## 2.10 Să utilizăm câteva clase Windows predefinite

Am văzut la paragraful 2.6 că parametrul `lpClassName` poate lua în funcția `CreateWindow()` câteva valori predefinite, fiind astfel create ferestre din clasele predefinite. Să adăugăm în fereastra noastră, câteva astfel de ferestre de clasă predefinită. Vom modifica fișierul *PrimaFereastra.cpp* ca mai jos:

```
int WINAPI WinMain( HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine,
                   int nCmdShow
                   )
{
    ...
    UpdateWindow( hwndMain );

    HWND hText = CreateWindow( "STATIC", "Eticheta Statica",
                               WS_CHILD | WS_VISIBLE | SS_LEFT, 20, 20, 100, 15,
                               hwndMain, NULL, hInstance, NULL );
    HWND hEdit = CreateWindow( "EDIT", "",
                               WS_CHILD | WS_VISIBLE | ES_LEFT | WS_BORDER,
                               20, 40, 100, 20, hwndMain, NULL, hInstance, NULL );
    HWND hBtn = CreateWindow( "BUTTON", "Buton de Comanda",
                              WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON,
                              20, 80, 150, 30, hwndMain, NULL, hInstance, NULL );
    HWND hListb = CreateWindow( "LISTBOX", "",
                                WS_CHILD | WS_VISIBLE | WS_DLGFRAE,
                                20, 120, 150, 50, hwndMain, NULL, hInstance, NULL );
    HWND hCombo = CreateWindow( "COMBOBOX", "",
                                WS_CHILD | WS_VISIBLE,
                                20, 180, 150, 30, hwndMain, NULL, hInstance, NULL );
    while ( GetMessage( &msg, (HWND) NULL, 0, 0 ) )
        ...
}
```

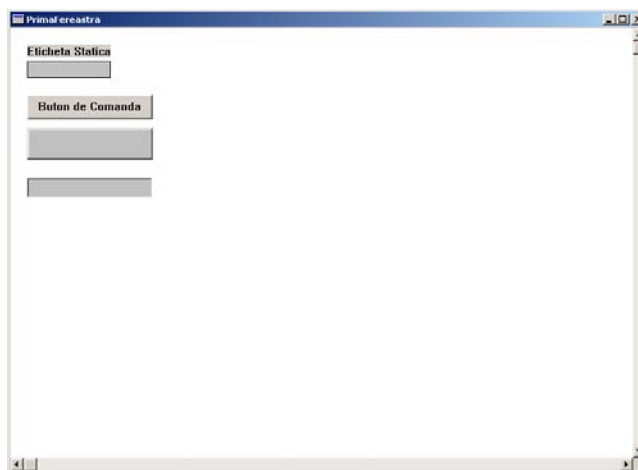


Figura 2.5. Interfața programului

Ce am făcut? Observăm că am utilizat funcțiile `CreateWindow()` cu denumirea clasei predefinite pentru fiecare fereastră în parte, asociind în același timp un identificator. Toate ferestrele sunt ferestre copil ale ferestrei principale, identificate prin identificatorul `hwndMain`. Acest identificator este transmis ca parametrul `hWndParent` funcției `CreateWindow()`. Am obținut astfel interfața din fig. 2.5.

## 2.11 Fișiere de resurse. Adăugarea unui meniu

Un program Windows, pe lângă fereastra principală poate conține și alte elemente: bare de instrumente, bare de stare, meniuri, diferite casete de dialog, etc. Aceste resurse în general nu se modifică în timpul execuției programului. Uzual, ele sunt descrise în fișiere text speciale, numite *fișiere de resurse*. Aceste fișiere au extensia *.rc*.

Din păcate, mediul Visual C++ afișează într-o formă grafică conținutul unui fișier de resurse, deci pentru a putea crea un astfel de fișier și a-l gestiona în format text, va trebui să “păcălim” mediul de programare.

Să creăm fișierul *PrimaFereastră.rc*. Vom crea un fișier gol cu acest nume, prin orice metodă cunoscută: *copy con*, *Notepad*, etc. Apoi, acest fișier va fi adăugat la proiect. Cum? Simplu. Alegem în meniu opțiunea **Project->Add to Project->Files...** și apoi, în browser se selectează fișierul *PrimaFereastră.rc*.

În acest fișier, vom descrie o bară de meniu asociată ferestrei programului. Înainte, să aflăm câte ceva despre meniuri.

Windows manipulează două tipuri de meniuri: *meniuri principale* (*top-level menu*) și respectiv *meniuri derulante* (*pop-up menu*). Meniul principal este constituit dintr-un set de comenzi vizibile permanent în bara de meniu a ferestrei principale. În marea majoritate a cazurilor, acestea reprezintă doar puncte de intrare pentru meniurile derulante. Aceste meniuri sunt desfășurate la alegerea unei opțiuni din meniul principal.

Să completăm fișierul *PrimaFereastră.rc* ca mai jos:

```
MENIUFEREASTRA MENU DISCARDABLE
BEGIN
    POPUP "&Casetă"
    BEGIN
        MENUITEM "&Start", IDM_START
        MENUITEM "&Arata", IDM_ARATA
        MENUITEM "Asc&unde", IDM_ASCUNDE
    END
    POPUP "&Gata"
    BEGIN
        MENUITEM "&Iesire", IDM_IESE
    END
END
```

Ce am făcut? Am declarat o resursă meniu (specificată prin cuvântul rezervat *MENU*) pe care o vom numi *MENIUFEREASTRA*. Prin specificarea faptului că această resursă este *DISCARDABLE* precizăm editorului de legături să elimine informația inițială a resursei despre meniu după ce programul înregistrează meniul în clasa ferestrei. Prin utilizarea acestui cuvânt rezervat pentru resursele meniu, se face o economie de memorie și viteza de execuție a programului crește.

Descriptorul *POPUP* precizează nivelul cel mai de sus al meniului (corespunzător în cazul nostru meniului principal). Linia *POPUP "&Casetă"* crează o intrare de forma casetă în meniul principal. Deci, caracterul „&” din definiție, va face ca litera care urmează să apară ca accelerator și va apare subliniată în meniu.

Prin alegerea acestei opțiuni, va fi deschisă lista derulantă care cuprinde toate elementele de meniu asociate intrării, descrise de descriptorul *MENUITEM* și cuprinse

între `BEGIN` și `END`. De exemplu, lista derulantă asociată intrării de meniu Caseta va conține intrările Start, Arata și Ascunde, identificate prin intermediul identificatorilor `IDM_START`, `IDM_ARATA` și `IDM_ASCUNDE`. Similar se crează intrarea de meniu Gata cu lista derulantă Iesire, identificată prin `IDM_IESE`.

Deoarece identificatorii sunt mărimi `UINT`, adică întregi fără semn, va trebui să asociem astfel de valori pentru identificatorii elementelor de meniu. Deoarece acești identificatori trebuie să fie recunoscuți în toate fișierele proiectului, ar fi bine să-i definim în fișierul *PrimaFereastră.h*:

```
#define IDM_START      110
#define IDM_ARATA      120
#define IDM_ASCUNDE    121
#define IDM_IESE       130
```

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

Valorile asociate identificatorilor sunt alese la întâmplare. Va trebui să includem fișierul header în fișierul resursă:

```
#include "PrimaFereastră.h"
MENIUFEREASTRA MENU DISCARDABLE
...
```

În acest moment, resursa meniu este definită și adăugată la proiect. Ea trebuie acum afișată de către fereastra principală a programului. Va trebui întâi să încărcăm resursa meniu și apoi să-l declarăm meniu implicit pentru fereastră.

Încărcarea meniului se face cu ajutorul funcției

```
HMENU LoadMenu(HINSTANCE hInstance, LPCTSTR lpMenuName);
```

cu semnificațiile parametrilor:

- `HINSTANCE hInstance` – identificatorul instanței programului în care se încarcă meniul;
- `LPCTSTR lpMenuName` – numele (din fișierul *.rc*) al meniului;

Meniul încărcat este declarat ca și meniu implicit al ferestrei programului, utilizând funcția:

```
BOOL SetMenu(HWND hWnd, HMENU hMenu);
```

unde

- `HWND hWnd` – identificatorul ferestrei în care se încarcă meniul;
- `HMENU hMenu` – identificatorul meniului (generat de funcția `LoadMenu()`);

Pentru încărcarea meniului, vom completa fișierul *PrimaFereastră.cpp* ca mai jos:

```
...
if (!hWndMain)
    return FALSE;
else
{
    HMENU hnewMenu=LoadMenu(hInstance, "MENIUFEREASTRA");
```



```

    SetMenu(hwndMain, hnewMenu);
}
...

```

Astfel meniul este încărcat numai dacă crearea ferestrei este reușită. În acest caz, obținem interfața din fig. 2.6.

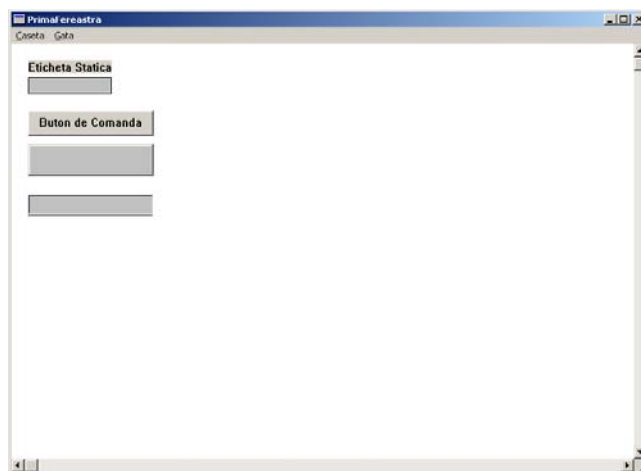


Figura 2.6. Interfața cu meniu

## 2.12 Interceptarea și prelucrarea evenimentelor generate de meniu

Orice selectare a unui element de meniu va face ca Windows să trimită mesajul `WM_COMMAND` buclei de mesaje. Identificatorul articolului de meniu este transmis ca și cuvântul cel mai puțin semnificativ al parametrului `wParam`. Deci, pentru identificarea intrării în meniu care a generat mesajul, funcția fereastră va trebui să testeze valoarea `loword(wParam)`.

Să modificăm programul astfel încât alegerea opțiunii **Iesire** să distrugă fereastra și să termine programul. Avem două posibilități de distrugere a ferestrei și de închidere a programului: fie să executăm aceeași secvență de instrucțiuni ca și în cazul mesajului `WM_CLOSE`, fie să utilizăm funcția de distrugere a ferestrei:

```
BOOL DestroyWindow(HWND hWnd);
```

unde

- `HWND hWnd` – identificatorul ferestrei care se distruge;

Pentru programul nostru, vom alege această din urmă variantă. Să ne reamintim că identificatorul elementului de meniu selectat este testat de secvența `loword(wParam)`. Va trebui să modificăm procedura fereastră ca mai jos:

```

LRESULT CALLBACK WndProc( HWND hwnd,
                          UINT uMsg,
                          WPARAM wParam,
                          LPARAM lParam
                        )
{
    switch(uMsg) {
        case WM_CLOSE:

```

```

    PostMessage(hwnd, WM_QUIT, 0L, 0L);
    return DefWindowProc(hwnd, uMsg, wParam, lParam);
case WM_KEYDOWN:
    LPCTSTR text;
    ...
    default: text="Ati apasat alta tasta";
            break;
    }
    MessageBox(hwnd, text, "Mesaj", MB_OK|MB_ICONEXCLAMATION);
    return DefWindowProc(hwnd, uMsg, wParam, lParam);
    break;
case WM_COMMAND :
    switch( LOWORD( wParam ) )
    {
        case IDM_IESE:
            DestroyWindow(hwnd);
            return DefWindowProc(hwnd, uMsg, wParam, lParam);
    }
default:
    return DefWindowProc(hwnd, uMsg, wParam, lParam);
}
}

```

Observăm că am transmis funcției `DestroyWindow()` identificatorul ferestrei principale. Aceasta va fi distrusă și programul se va termina.

Să modificăm acum programul, astfel încât ferestrele de clase predefinite să nu fie create și să apară la lansarea în execuție a programului, ci doar la selecția opțiunii de meniu **Arata**. De asemenea, la alegerea opțiunii **Ascunde**, aceste ferestre dorim să fie distruse.

Pentru aceasta, în primul rând, va trebui să ștergem liniile de program introduse la paragraful 2.10, deoarece acele ferestre sunt create doar în cadrul procedurii fereastră, la prelucrarea mesajului `WM_COMMAND` generat de `IDM_ARATA`.

Apoi, va trebui să facem observația că procedura fereastră nu primește ca parametru instanța programului căruia îi aparține fereastra principală, dar la crearea instanțelor ferestrelor predefinite este nevoie de aceasta. Deci, va trebui să declarăm o variabilă `HINST` globală, care după crearea ferestrei principale să se încarce cu identificatorul instanței utilizat de clasa fereastră (`hInstance` în cazul nostru). Această variabilă globală va putea fi utilizată apoi de către procedura fereastră. Pentru aceasta, vom modifica fișierul *PrimaFereastră.cpp* ca mai jos:

```

#include <Windows.h>
#include "PrimaFereastra.h"
HINSTANCE hInst;
int WINAPI WinMain( HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPSTR lpCmdLine,
    int nCmdShow
)
{
    ...
    hwndMain = CreateWindow("MainWndClass", "PrimaFereastra",
        WS_OVERLAPPEDWINDOW | WS_HSCROLL | WS_VSCROLL, CW_USEDEFAULT,
        CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, (HWND) NULL,
        (HMENU) NULL, hInstance, (LPVOID) NULL);
    hInst=hInstance;
    if (!hwndMain)
        ...

```

```
}
```

Modificările în funcția fereastră vor fi:

```
LRESULT CALLBACK WndProc( HWND hwnd,
                          UINT uMsg,
                          WPARAM wParam,
                          LPARAM lParam
                          )
{
    static HWND hText=NULL;
    static HWND hEdit=NULL;
    static HWND hBtn=NULL;
    static HWND hListb=NULL;
    static HWND hCombo=NULL;

    switch(uMsg){
    case WM_CLOSE:
        ...
    case WM_COMMAND :
        switch( LOWORD( wParam ) )
        {
            case IDM_ARATA:
                hText = CreateWindow( "STATIC", "Eticheta Statica",
                                     WS_CHILD | WS_VISIBLE | SS_LEFT,20, 20, 100, 15,
                                     hwnd,NULL, hInst, NULL );
                hEdit = CreateWindow( "EDIT", "",WS_CHILD | WS_VISIBLE |
                                     ES_LEFT | WS_BORDER,20, 40, 100, 20,
                                     hwnd,NULL,hInst, NULL );
                hBtn = CreateWindow( "BUTTON", "Buton de Comanda",
                                    WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON,
                                    20, 80, 150, 30, hwnd,NULL, hInst, NULL );
                hListb = CreateWindow( "LISTBOX", "",
                                     WS_CHILD | WS_VISIBLE | WS_DLGFRAME ,
                                     20, 120, 150, 50, hwnd,NULL, hInst, NULL );
                hCombo = CreateWindow( "COMBOBOX", "",
                                     WS_CHILD | WS_VISIBLE , 20, 180, 150, 30,
                                     hwnd,NULL, hInst, NULL );
                break;
            case IDM_ASCUNDE:
                DestroyWindow(hText);
                DestroyWindow(hEdit);
                DestroyWindow(hBtn);
                DestroyWindow(hListb);
                DestroyWindow(hCombo);
                break;
            case IDM_IESE:
                DestroyWindow(hwnd);
                return DefWindowProc(hwnd, uMsg, wParam, lParam);
        }
    default:
        return DefWindowProc(hwnd, uMsg, wParam, lParam);
    }
}
```

Să vedem ce am modificat. În primul rând, am declarat identificatori pentru fiecare din obiectele create de apelul funcției `CreateWindow()`. Acești identificatori trebuie declarați la începutul procedurii fereastră, deoarece ei trebuie să fie vizibili în corpul a doi selectori `case`. Am declarat de asemenea variabilele ca fiind statice pentru ca domeniul lor de existență să fie toată durata programului. Cum

identificatorii sunt variabile statice, ei trebuie inițializați și cum nu sunt asociați nici unei ferestre deocamdată, e normal să fie inițializați cu `NULL`. La apelul funcțiilor `CreateWindow()`, ca deosebire față de paragraful 2.10, se pot enumera utilizarea parametrului `hwnd` pentru identificarea instanței ferestrei principale și respectiv identificatorul global `hInst`, în loc de `hwndMain` și respectiv `hInstance`, al căror domeniu de vizibilitate se rezumă la funcția `WinMain()`.

### 2.13 Casete de dialog.

Casetele de dialog sunt un alt tip de fereastră foarte des utilizată de programele Windows pentru implementarea interfețelor. Aceste casete sunt utilizate de obicei pentru a prezenta de o manieră unitară diferite controale necesare introducerii și afișării datelor. Spre deosebire de celelalte tipuri de ferestre, casetele de dialog folosesc o funcție implicită de prelucrare a mesajelor de la tastatură, făcând mai ușoară prelucrarea intrărilor furnizate de utilizator. Casetele de dialog pot fi afișate ca și fereastră principală a programului, dar, în foarte multe programe ele sunt folosite ca și ferestre copil, lansate la alegerea unor opțiuni de meniu în fereastra principală a programului. Acest mod de utilizare a lor îl vom descrie în cele ce urmează.

Casetele de dialog pot fi afișate în două moduri: ca și casete **modale** și respectiv, ca și casete **nemodale**. O casetă modală acaparează în totalitate controlul mesajelor, astfel că utilizatorul nu va putea face activă o altă fereastră a programului care a lansat caseta, până la închiderea casetei modale. Utilizatorul va putea însă comuta în orice altă fereastră aparținând altui program. Ca o consecință, programul care a lansat caseta modală nu poate comanda închiderea acesteia, acest lucru putându-se face doar din interiorul casetei.

Casetele nemodale pot primi sau pierde focusul fără nici o problemă. Spre deosebire de casetele modale, ele pot fi închise atât din interiorul lor, cât și din orice altă fereastră a programului din care fac parte.

Înainte de utilizare, caseta de dialog trebuie construită. Construcția ei este realizată pe baza unui șablon, implementat uzual în fișierul de resurse. Șablonul are următoare construcție:

```
identificator_caseta DIALOG DISCARDABLE x, y, lungime, inaltime
STYLE stil1 | stil2 | ... | stiln
CAPTION "Titlul casetei de dialog"
FONT dimensiune_litere, „nume litere”
BEGIN
    descriere control1
    ...
    descriere controln
END
```

Mărimile `x`, `y`, `lungime`, `inaltime` reprezintă coordonatele colțului stânga sus a casetei, respectiv dimensiunile ei. Controalele descrise în secțiunea `BEGIN ... END` a șablonului pot fi alese din următoarele: `BUTTON`, `CHECKBOX`, `COMBOBOX`, `CONTROL`, `CTEXT`, `DEFPUSHBUTTON`, `EDITTEXT`, `GROUPBOX`, `ICON`, `LISTBOX`, `LTEXT`, `PUSHBUTTON`, `RADIOBUTTON`, `RTEXT`, `SCROLLBAR`, `STATIC`.

Să modificăm fișierul *PrimaFereastră.rc* astfel încât să descriem o casetă de dialog:

```
#include <Windows.h>
```

```
#include "PrimaFereastra.h"

MENIU MENU DISCARDABLE
BEGIN
    POPUP "&Casetă"
    BEGIN
        MENUITEM "&Start", IDM_START
        MENUITEM "&Arată", IDM_ARATA
        MENUITEM "Asc&unde", IDM_ASCUNDE
    END
    POPUP "&Gata"
    BEGIN
        MENUITEM "&Iesire", IDM_IESE
    END
END

CASETAMODALA DIALOG DISCARDABLE 22, 17, 200, 140
STYLE DS_MODALFRAME | WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU
CAPTION "Casetă Modala/Nemodala"
FONT 10, "Times New Roman"
BEGIN
    CONTROL "OK", IDOK, "BUTTON", BS_DEFPUSHBUTTON | WS_CHILD
        | WS_VISIBLE | WS_GROUP | WS_TABSTOP, 140, 10, 50, 14
    CONTROL "Cancel", IDCANCEL, "BUTTON", BS_DEFPUSHBUTTON | WS_CHILD
        | WS_VISIBLE | WS_GROUP | WS_TABSTOP, 140, 30, 50, 14
    GROUPBOX "Butoane Radio", IDC_GRPB, 10, 10, 100, 50
    RADIOBUTTON "Adevarat", IDC_RADIO1, 30, 25, 110, 10, WS_GROUP
        | WS_TABSTOP
    RADIOBUTTON "Fals", IDC_RADIO2, 30, 40, 110, 10, WS_TABSTOP
    CONTROL "Introduceți textul", -1, "STATIC", SS_LEFT | WS_CHILD
        | WS_VISIBLE | WS_GROUP, 15, 100, 80, 10
    EDITTEXT IDC_EDIT1, 90, 95, 100, 20
END
```

Ce descrie șablonul? O casetă de dialog identificată de identificatorul CASETAMODALA, care are colțul stânga sus la coordonatele (22,17) și dimensiunile 200 respectiv 140 pixeli. Casetă de dialog va avea titlul în bara de titlu "*Casetă Modala/Nemodala*" și va utiliza litere Times New Roman de dimensiune 10. Casetă va conține un buton de comandă cu eticheta *OK*, având identificatorul IDOK, cu colțul stânga sus de coordonate (140,10) și dimensiuni 50 și 14, etc.

Se observă că descrierea controalelor afișate de casetă se poate face în două moduri: cu cuvântul `CONTROL` și respectiv doar cu tipul controlului. În descrierea de mai sus am folosit ambele moduri de descriere, din rațiuni didactice, dar uzual, se utilizează doar unul din ele. Cele două moduri de descriere sunt absolut echivalente și le puteți găsi bine documentele în MSDN.

Deoarece șabloanele de descriere a controalelor sunt stocate în *Windows.h*, va trebui să includem și această bibliotecă. De asemenea, vor trebui asociate valori în fișierul *PrimaFereastra.h* pentru identificatorii nou introduși:

```
...
#define IDM_IESE 130
#define IDC_GRPB 201
#define IDC_RADIO1 202
#define IDC_RADIO2 203
#define IDC_EDIT1 204
```

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

Am obținut astfel o casetă de dialog cu aspectul din fig. 2.7.

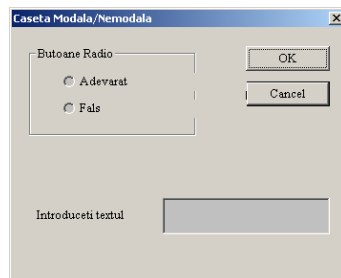


Figura 2.6. Casetă de dialog

### 2.13.1 Afișarea modală a casetei de dialog

Casetă de dialog e creată, dar programul nu o afișează. Dorim ca afișarea ei să se facă modal, la alegerea opțiunii **Start**. Lansarea modală a unei casete de dialog se face cu macrocomanda (similară funcției)

```
int DialogBox(HINSTANCE hInstance, LPCTSTR lpTemplate,
             HWND hWndParent, DLGPROC lpDialogFunc );
```

cu parametrii:

- `HINSTANCE hInstance` – identificatorul instanței programului în care se lansează caseta;
- `LPCTSTR lpTemplate` – numele sau identificatorul (din fișierul `.rc`) al șablonului ce descrie caseta de dialog;
- `HWND hWndParent` – identificatorul ferestrei părinte (din care se lansează caseta);
- `DLGPROC lpDialogFunc` – un pointer spre o funcție callback care tratează mesajele transmise spre casetă (procedura casetă de dialog);

În concluzie, va trebui să modificăm procedura fereastră, astfel încât la mesajul `WM_COMMAND` având ca parametru `IDM_START` să lanseze modal caseta de dialog:

```
LRESULT CALLBACK WndProc( HWND hwnd,
                          UINT uMsg,
                          WPARAM wParam,
                          LPARAM lParam
                          )
{
    ...
    case IDM_IESE:
        DestroyWindow(hwnd);
        return DefWindowProc(hwnd, uMsg, wParam, lParam);
    case IDM_START :
        DialogBox( hInst, "CASETAMODALA", hwnd, (DLGPROC)Modal );
        break;
    }
    default:
        return DefWindowProc(hwnd, uMsg, wParam, lParam);
}
```

```
}
```

Observăm că parametrul `DLGPROC lpDialogFunc` conține adresa unei funcții numite `Modal()`. Aceasta este funcția `callback` (pe care urmează să o declarăm și să o definim) care va trata mesajele asociate casetei de dialog. De ce e nevoie de o conversie explicită de tip? Să ne reamintim, funcțiile `callback` sunt de tip `LRESULT`, deci va trebui să facem explicit o conversie la tipul `DLGPROC`.

Acum va trebui să declarăm funcția, ca o funcție `callback` obișnuită. O vom face în fișierul *PrimaFereastră.h*:

```
...
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK Modal(HWND, UINT, WPARAM, LPARAM);
```

Definirea funcției o vom face în fișierul *PrimaFereastră.cpp*, ca mai jos:

```
LRESULT CALLBACK WndProc( HWND hwnd,
                          UINT uMsg,
                          WPARAM wParam,
                          LPARAM lParam
                          )
{
    ...
    default:
        return DefWindowProc(hwnd, uMsg, wParam, lParam);
}

LRESULT CALLBACK Modal( HWND hDlg,
                        UINT message,
                        WPARAM wParam,
                        LPARAM lParam)
{
    switch (message)
    {
    case WM_INITDIALOG:
        return (TRUE);
    case WM_COMMAND:
        if (LOWORD(wParam) == IDOK
            || LOWORD(wParam) == IDCANCEL)
        {
            EndDialog(hDlg, TRUE);
            return (TRUE);
        }
        break;
    }
    return (FALSE);
}
```

Funcția tratează două mesaje: primul, `WM_INITDIALOG`, este mesajul transmis de Windows procedurii dialog chiar înainte de afișarea casetei. Este utilizat de obicei pentru inițializarea controalelor conținute de casetă. În cazul nostru, nu face altceva decât să marcheze faptul că crearea și afișarea casetei de dialog a reușit, dar în capitolele următoare ne va fi deosebit de util. Al doilea mesaj tratat este `WM_COMMAND`, care tratează apăsarea butoanelor `OK` și `Cancel`. Identificatorii acestora sunt transmiși

în cadrul mesajului în partea cea mai puțin semnificativă a parametrului `wParam`. La apăsarea unuia din butoane, se apelează funcția:

```
BOOL EndDialog(HWND hDlg, int nResult);
```

unde

- `HWND hDlg` – identificatorul casetei de dialog căreia îi este asociată procedura dialog;
- `int nResult` – valoarea returnată aplicației de funcția ce a creat caseta de dialog;

Astfel, la apăsarea unuia din cele 2 butoane, caseta va fi închisă.

### 2.13.2 Afișarea nemodală a casetei de dialog

O casetă este afișată nemodal prin intermediul macrocomenzii

```
HWND CreateDialog(HINSTANCE hInstance, LPCTSTR lpTemplate,
    HWND hWndParent, DLGPROC lpDialogFunc);
```

cu

- `HINSTANCE hInstance` – un identificator al instanței programului care conține șablonul de creare a casetei de dialog;
- `LPCTSTR lpTemplate` – numele sau identificatorul (din fișierul `.rc`) al șablonului ce descrie caseta de dialog;
- `HWND hWndParent` – identificatorul ferestrei părinte (din care se lansează caseta);
- `DLGPROC lpDialogFunc` – un pointer spre o funcție callback care tratează mesajele transmise spre casetă (procedura casetă de dialog);

Funcția returnează în caz de succes un identificator al casetei de dialog create. Pentru a fi vizibilă, caseta trebuie neapărat să aibă stilul `WS_VISIBLE`, spre deosebire de caseta modală, de a cărei afișare este responsabilă macrocomanda `DialogBox()`. O casetă creată cu macrocomanda `CreateDialog()` va trebui la închidere distrusă cu funcția `DestroyWindow()`.

Să afișăm acum nemodal caseta de dialog. Va trebui întâi să adăugăm stilul `WS_VISIBLE` la șablonul care descrie caseta de dialog:

```
CASETAMODALA DIALOG DISCARDABLE 22, 17, 200, 140
STYLE DS_MODALFRAME | WS_OVERLAPPED | WS_CAPTION
    | WS_SYSMENU | WS_VISIBLE
...
```

Acest nou stil adăugat nu influențează cu nimic afișarea modală a casetei. Acum, va trebui să creem o nouă intrare în meniu, care să permită lansarea nemodală. Fie **Deschide Nemodal** această intrare. Pentru aceasta, va trebui să modificăm fișierul de resurse,

```
...
MENIU MENU DISCARDABLE
```



```

BEGIN
    POPUP "&Caseta"
    BEGIN
        ...
        MENUITEM "Asc&unde", IDM_ASCUNDE
        MENUITEM "&Deschide Nemodal", IDM_DNEMOD
    END
    ...

```

și să definim în fișierul header noul identificator adăugat:

```

...
#define IDM_IESE 130
#define IDM_DNEMOD 140
...

```

Va trebui acum, să declarăm în fișierul sursă, un identificator `HWND` care să fie actualizat de macrocomanda `CreateDialog()` în caz de succes. Deoarece acest identificator va trebui să fie vizibil atât în procedura fereastră, care lansează nemodal caseta, cât și în procedura dialog asociată casetei, vom declara global acest identificator:

```

#include <Windows.h>
#include "PrimaFereastra.h"

HINSTANCE hInst;
HWND pDlgNemod=NULL;
...

```

Nu ne mai rămâne altceva de făcut, decât să interceptăm în cadrul procedurii fereastră a mesajului `WM_COMMAND` generat de alegerea intrării de meniu **Deschide Nemodal** (având identificatorul `IDM_DNEMOD`):

```

LRESULT CALLBACK WndProc( HWND hwnd,
                          UINT uMsg,
                          WPARAM wParam,
                          LPARAM lParam
                          )
{
    ...
    case IDM_START :
        DialogBox( hInst, "CSETAMODALA", hwnd, (DLGPROC)Modal );
        break;
    case IDM_DNEMOD:
        if (!pDlgNemod)
            pDlgNemod=CreateDialog(hInst, "CSETAMODALA",
                                hwnd, (DLGPROC)Nemodal);
            break;
        }
    default:
        return DefWindowProc(hwnd, uMsg, wParam, lParam);
    }
}

```

Cum funcționează? Dacă `pDlgNemod` e `NULL`, adică nu este încărcat cu adresa nici unei casete nemodale, se apelează macrocomanda `CreateDialog()`. Dacă aceasta se execută cu succes, va actualiza identificatorul `pDlgNemod` cu adresa casetei

nemodale. Astfel, va exista la un moment dat o singură casetă modală lansată, crearea și afișarea unei noi casete nemodale fiind împiedicată de condiția `if()`. Macrocomanda `CreateDialog()` primește ca ultim argument un pointer spre o procedură dialog care tratează mesajele generate de Windows spre casetă. Această procedură dialog am numit-o `Nemodal()` și va trebui declarată și definită.

Pentru aceasta, în fișierul *PrimaFereastră.h* vom adăuga linia

```
...
LRESULT CALLBACK Modal(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK Nemodal(HWND, UINT, WPARAM, LPARAM);
```

iar în fișierul *PrimaFereastră.cpp* vom defini funcția ca mai jos:

```
LRESULT CALLBACK Nemodal( HWND hDlg,
                        UINT message,
                        WPARAM wParam,
                        LPARAM lParam)
{
    switch (message)
    {
        case WM_INITDIALOG:
            return (TRUE);
        case WM_COMMAND:
            if ( LOWORD(wParam) == IDOK
                || LOWORD(wParam) == IDCANCEL)
            {
                DestroyWindow(hDlg);
                pDlgNemod=NULL;
                return (TRUE);
            }
            break;
        }
    return (FALSE);
}
```

La apăsarea butoanelor `IDOK` sau `IDCANCEL`, este apelată funcția `DestroyWindow()`, care distruge caseta nemodală. Apoi, identificatorul `pDlgNemod` este încărcat cu `NULL`, pentru a putea fi lansată o nouă casetă de dialog.

Spre deosebire de casetele modale, o casetă nemodală poate fi închisă chiar de programul care a lansat-o. Este oarecum normal, dacă ne gândim că o astfel de casetă nu capturează în totalitate controlul interfeței ca și o casetă modală. Haideți să încercăm să implementăm și această activitate. Pentru început, în fișierul *PrimaFereastră.rc* să declarăm o nouă intrare de meniu:

```
...
MENIU MENU DISCARDABLE
BEGIN
    POPUP "&Casetă"
    BEGIN
        ...
        MENUITEM "&Deschide Nemodal", IDM_DNEMOD
        MENUITEM "&Inchide Nemodal", IDM_INEMOD
    END
END
...
```

Acum, după cum deja știm, va trebui să asociem o valoare noului identificator (în fișierul `PrimaFereastră.h`):

```
...
#define IDM_DNEMOD 140
#define IDM_INEMOD 141
...
```

Și, în final, să tratăm mesajul corespunzător în procedura fereastră:

```
case IDM_DNEMOD:
    ...
    break;
case IDM_INEMOD:
    if (pDlgNemod)
    {
        DestroyWindow(pDlgNemod);
        pDlgNemod=NULL;
    }
    break;
...
```

Observăm că modul de distrugere a casetei de dialog este similar cu cel din funcția `Nemodal()`. De această dată însă, funcția `DestroyWindow()` primește ca parametru identificatorul `pDlgNemodal`. E corect, acesta identifică caseta nemodală.

Ca o observație, vom putea lansa simultan caseta, atât modal cât și nemodal (fig. 2.7):

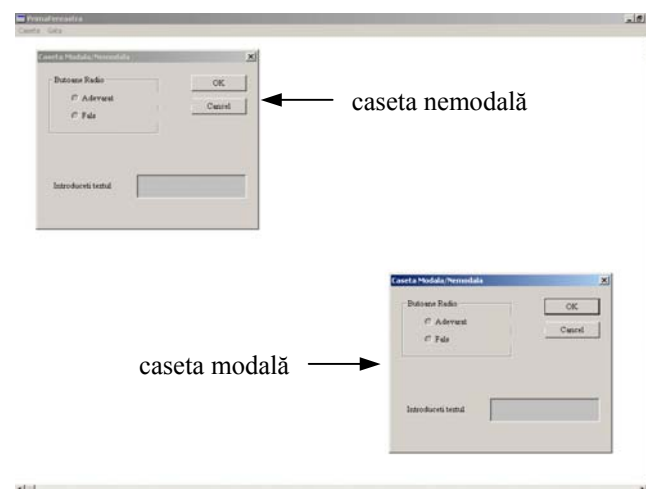


Figura 2.7. Se pot lansa simultan o casetă modală și una nemodală

## Întrebări și probleme propuse

1. Implementați și executați toate exemplele propuse în capitolul 2;
2. Cum se transmite o acțiune a utilizatorului către un program aflat în execuție?
3. Ce este un indentificator? De ce tip este identificatorul unei ferestre?
4. Adăugați programului dumneavoastră următoarea secvență de cod aferentă ferestrei principale:

```
case WM_CLOSE:
```

```
MessageBox(hwnd, "Aplicatie terminata!", "Mesaj",  
            MB_OK|MB_ICONEXCLAMATION);  
PostMessage(hwnd, WM_QUIT, 0L, 0L);  
return DefWindowProc(hwnd, uMsg, wParam, lParam);
```

Unde trebuie adăugat și ce efect are?

5. Modificați programul astfel încât fereastra principală a programului să nu mai aibă butoanele de minimizare și maximizare și nici barele de defilare orizontală și verticală.
6. La ce folosesc fișierele de resurse?
7. De ce identificatorul `pDlgNemod` este definit global?