

9. Utilizarea claselor de fișiere secvențiale

Marea majoritate a aplicațiilor trebuie să păstreze un timp îndelungat, de obicei pe un suport magnetic, datele de intrare, sau rezultatul execuției lor, sub forma unor fișiere. Cunoaștem că ANSI C pune la dispoziția utilizatorilor diferite funcții pentru manipularea fișierelor, furnizate de biblioteca `<stdio.h>`, cum ar fi: `fopen()`, `fclose()`, `fprintf()`, `fscanf()`, `fflush()`, etc. Aceste funcții construiesc sau utilizează un identificator atașat fișierului cu care se lucrează, toate operațiile asupra fișierului facându-se pe baza acestui identificator. De asemenea, apariția unor erori în manipularea fișierelor, este semnalizată tot de către acest identificator, fiind din păcate, mai dificil de determinat cauza care a produs eroarea.

Aceste funcții pot fi utilizate și în programele a căror interfață este construită cu MFC, dar acest lucru nu este de dorit. MFC pune la dispoziție o ierarhie de clase pentru lucrul cu fișiere, mult mai ușor de utilizat și mult mai clare în detectarea și identificarea erorilor.

Înainte de a prezenta care sunt aceste clase și cum se lucrează cu ele, să mai studiem o tehnică de interceptare a erorilor, deosebit de des utilizată în limbajele moderne, cum ar fi C++ sau Java.

9.1 Excepții

Complexitatea actuală a aplicațiilor software, inclusiv a celor orientate pe obiecte, presupune o modularizare pe nivele ierarhice. Această situație impune tratarea deosebit de riguroasă a erorilor și a situațiilor deosebite care pot să apară la nivelul diferitelor componente ale unui proiect.

În cele ce urmează, vom înțelege prin excepție o situație deosebită care poate să apară pe parcursul execuției unei componente soft parte a unei aplicații. Erorile pot fi privite ca și un caz particular de excepții, dar în general nu toate excepțiile sunt erori.

Excepțiile se împart în două categorii: excepții *sincrone* și excepții *asincrone*. Excepțiile sincrone sunt excepții a căror apariție poate fi prevăzută de programator (de exemplu, un fișier care se dorește a fi prelucrat nu se află în directorul așteptat). Excepțiile asincrone sunt excepții a căror apariție nu poate fi prevăzută în momentul implementării programului (de exemplu, defectarea accidentală a hard-disk-ului). Mecanismul de tratare a excepțiilor ia în considerare numai situația excepțiilor sincrone.

Problema se pune astfel: în faza de programare, se poate presupune că un modul de program va detecta în anumite situații o excepție, dar nu va putea oferi o soluție generală de tratare a acesteia, în timp ce un alt modul al aplicației poate oferi o soluție de tratare a excepției, dar nu poate detecta singur apariția excepțiilor. Un exemplu concret al unui astfel de caz este următorul: presupunem că avem o aplicație care lucrează cu fișiere. Un modul al aplicației (numit în cele ce urmează sursă) declară o clasă `CFile` care poate detecta o excepție din clasa *fișier inexistent*. Această clasă însă nu poate să întreprindă o acțiune generală de tratare a excepției prinse. Tratarea excepției se face în funcție de modulul (numit destinație) care folosește clasa `CFile`. Un modul ar putea, de exemplu, să invoce funcția `exit()` și să termine aplicația, în

timp ce un alt modul ar putea doar să afișeze o casetă de dialog și să solicite utilizatorului selectarea unui alt fișier.

Soluția obiectuală a acestei probleme este bazată pe următoarea idee fundamentală: modulul care detectează excepția pe care nu o poate rezolva o pasează (**throw**) modulului client, în speranța că o componentă a acestuia o va prinde (**catch**) și o va trata.

9.1.1 Metode neobiectuale de tratare a excepțiilor

Următoarele metode sunt folosite în mod tradițional pentru tratarea excepțiilor:

- terminarea aplicației - este o soluție folosită în special de sistemul de operare, în cazul în care o excepție aruncată nu este prinsă;
- returnarea unui cod de eroare;
- poziționarea unui indicator global;
- ignorarea excepției - este o soluție periculoasă. În cazul în care excepția nu este o eroare, aplicația poate să mai ruleze până când se produce o eroare. Din păcate, este foarte greu de depanat și de găsit eroarea în astfel de situații, deoarece nu se poate detecta momentul producerii ei.
- invocarea unei funcții specializate de tratare a excepției, numită *handler*;

9.1.2 Mecanismul C++ de tratare a excepțiilor

Alternativa oferită de limbajul C++ pentru tratarea excepțiilor furnizează o soluție îmbunătățită, simplificând cooperarea între componente scrise de mai mulți programatori, dar integrate în aceeași aplicație.

Implementarea C++ a mecanismului de tratare a excepțiilor are la bază următoarele trei cuvinte cheie:

- **try** – care delimitează la nivelul modulului destinație codul care ar putea să emită (raise) o excepție;
- **throw** - care se execută în cazul în care modulul sursă a detectat o condiție de excepție;
- **catch** – delimitează o secvență de cod la nivelul modulului destinație care va trata o situație excepțională în cazul detectării ei;

Așadar, tratarea excepțiilor printr-un mecanism C++ se face în următoarele trei etape:

- se emite excepția prin **throw**;
- se detectează excepția prin **try**;
- se tratează excepția prin **catch**;

Forma generală a secvenței de program ce tratează excepțiile este

```
try {  
    ...  
    if (eroare) throw excepție_de_tipul_stabilit;  
    ...  
}
```

```
...
catch (tip_excepție* variabilă) {
    // prelucrarea excepției
}
```

Un exemplu banal de tratare a unei excepții este prezentat în continuare: să presupunem că scriem următorul program:

```
#include <iostream.h>
double impart(double deimpartit, double impartitor)
{
    return deimpartit/impartitor;
}

void main(void)
{
    double de, im;
    cout << "\n Deimpartit= ";
    cin >> de;
    cout << "\n Impartitor= ";
    cin >> im;
    cout << "\n Rezultat= " << impart(de,im) << " ";
}
```

Este evident, programul așteaptă două valori reale și afișează rezultatul împărțirii lor. Dar, cum cele 2 valori sunt externe programului, furnizate de utilizator, nimic nu-l împiedică pe acesta să dea împărțitorului valoarea 0, ajungând în situația anormală de împărțire a unui număr la 0. Astfel de situații de *excepție* sunt uzual tratate de sistemele de operare prin terminarea forțată a programelor. O astfel de situație este prezentată în fig. 9.1 a și b, respectiv acțiunile efectuate de *MsDOS* și *Windows*, în cazul programelor scrise în *Borland C* și *Visual C++* într-un proiect de tip Win32 Console Application.

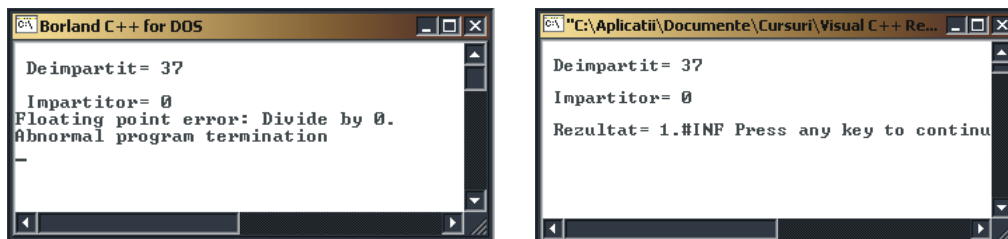


Figura 9.1. Terminarea forțată a programului în MsDOS și Windows

Se poate observa că în ambele cazuri, sistemul de operare termină forțat programul. Putem face prin mecanismul `try - throw - catch`, ca eroarea să nu fie pasată sistemului de operare, ci reținută în cadrul programului, pentru a fi tratată de acesta. Pentru aceasta, să modificăm programul ca mai jos:

```
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>

double impart(double deimpartit, double impartitor)
{
    if (impartitor==0) throw "Eroare: impartire la 0";
    return deimpartit/impartitor;
}
```

```

void main(void)
{
    double de, im;
    cout << "\n Deimpartit= ";
    cin >> de;
    while (1)
    {
        cout << "\n Impartitor= ";
        cin >> im;
        try
        {
            cout << "\n Rezultat= " << impart(de,im) << " ";
            exit(1);
        }
        catch (char* exceptie)
        {
            cout << "\n " << exceptie << " ";
        }
    }
}

```

Ce observăm? În caz de funcționare normală, programul afișează rezultatul împărțirii și se termină, datorită instrucțiunii `exit(1)`. În caz de împărțire cu 0, programul se reia ciclic, semnalând împărțirea cu 0 și cerând un nou împărțitor. Deci, programul nu mai execută instrucțiunea următoare apariției excepției (aruncată de funcția `impart()`), adică `exit(1)`, ci sare în blocul `catch`, execută secvența de tratare a excepției din acesta și continuă cu secvența de program de după blocul `catch`. (fig. 9.2).

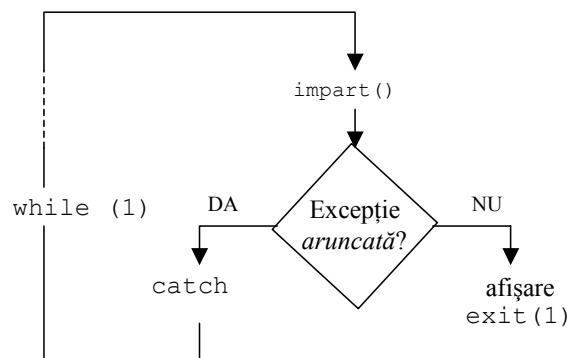


Figura 9.2. Execuția secvenței try - catch

Se impune și o a doua remarcă: excepția trebuie considerată ca fiind o instanțiere a unui obiect! În exemplul de mai sus, excepția este un obiect de un tip predefinit, anume șir de caractere. Dacă excepția emisă nu ar fi fost “prinsă” în blocul `catch`, sistemul de operare ar fi intervenit și ar fi stopat execuția programului.

Excepțiile pot să apară din diferite cauze, deci și tipurile excepțiilor lansate în blocurile `try` pot fi diferite. Este absolut necesar ca excepția capturată de blocul `catch` să fie de același tip cu cea lansată anterior în blocul `try`. De menționat faptul că un bloc `try` trebuie să fie urmat de unul sau mai multe blocuri `catch`, câte unul pentru fiecare tip de excepție care s-ar putea emite în blocul `try`. Exemplul de mai jos prezintă această situație:

```

#include <iostream.h>
#include <string.h>

// clasa Exceptie implementeaza un obiect exceptie, care contine
// informatie despre tipul exceptiei

class ExceptieObiectuala
{
public:
    char* tip_exceptie;
    ExceptieObiectuala(char* arg=NULL);
    ~ExceptieObiectuala();
};

ExceptieObiectuala::ExceptieObiectuala(char* arg)
{
    if(arg!=NULL)
    {
        tip_exceptie = new char[strlen(arg)+1];
        strcpy(tip_exceptie, arg);
    }
    else
        tip_exceptie = NULL;
}

ExceptieObiectuala::~~ExceptieObiectuala()
{
    if(tip_exceptie!=NULL)
        delete tip_exceptie;
}

// clasa care emite exceptii..
class Test
{
public:
    // emite o exceptie de tip string
    void ThrowString();

    //emite o exceptie de tip Exception
    void ThrowExceptieObiectuala();
};

void Test::ThrowString()
{
    throw "EXCEPTIE TIP SIR DE CARACTERE!!";
}

void Test::ThrowExceptieObiectuala()
{
    throw *new
        ExceptieObiectuala("EXCEPTIE DE CLASA ExceptieObiectuala!!");
}

void main()
{
    Test t;

    // emite exceptie de tip string
    try{
        t.ThrowString();
    }
}

```

```

catch(char* sir1_Exceptie){
    cout<<sir1_Exceptie<<"\n";
}
catch(ExceptieObiectuala& ref_exceptie1){
    cout<<ref_exceptie1.tip_exceptie<<"\n";
}

// asa se trateaza exceptiile neinterceptate în alte blocuri ...
catch(...)
{
    ;
}

// emite un obiect exceptie de clasa ExceptieObiectuala
try{
    t.ThrowExceptieObiectuala();
}
catch(char* sir2_Exceptie)
{
    cout<< sir2_Exceptie <<"\n";
}
catch(ExceptieObiectuala& ref_exceptie2)
{
    cout<< ref_exceptie2.tip_exceptie <<"\n";
}
}

```

Ce se poate remarca din acestui exemplu? În primul rând, modul de tratare a excepțiilor care nu sunt prinse în alte blocuri `catch`, cu construcția `catch(...)`. În al doilea rând, blocurile `catch` prind o referință la un obiect `ExceptieObiectuala` și nu un obiect `ExceptieObiectuala` transmis prin valoare. Aceasta este necesar deoarece, în cazul transmiterii unui obiect `ExceptieObiectuala` prin valoare, s-ar apela constructorul de copiere (furnizat de compilator), cu efecte dezastruoase!

9.2 Clasele MFC pentru lucrul cu fișiere

MFC furnizează două clase pentru lucrul cu fișiere standard. Clasa `CFile` permite accesul la fișiere binare stocate pe disc. Clasa încapsulează un identificator pentru accesul la fișiere și oferă metode de deschidere, citire, scriere și închidere a fișierelor. Clasa `CFile` realizează un acces direct la fișier, fără stocarea prealabilă într-o zonă tampon (buffer) a datelor.

Clasa `CStdioFile`, derivată din clasa `CFile`, implementează operații cu fișiere în flux. În acest caz, datele sunt în prealabil stocate în buffere.

Tot derivate din clasa `CFile` sunt și clasa `CMemFile` care asigură suport pentru fișierele stocate în memoria RAM, cum ar fi cele salvate pe RAM-disk-uri, utilizate pentru creșterea vitezei de acces la date, sau clasa `CSharedFile`, care asigură suport pentru accesul la fișiere partajate, aflate în memorie.

9.2.1 Deschiderea fișierelor

Clasa `CFile` permite deschiderea unui fișier pe mai multe căi. Câteva din acestea sunt prezentate mai jos:

- clasa **CFile** furnizează un constructor care permite specificarea fişierului de pe disc care urmează a fi deschis. Acest constructor permite declararea unui obiect **CFile** şi îl asociază cu un fişier existent. Cum deschiderea fişierului se face în constructorul clasei, o situaţie de incident, cum ar fi inexistenţa fişierului specificat, poate duce la erori fatale;
- o cale mai normală presupune deschiderea fişierului în doi paşi. Întâi se creează un obiect **CFile**, apoi acestui obiect i se asociază un fişier de pe disc. Pentru deschiderea fişierului, se utilizează funcţia

```
virtual BOOL Open( LPCTSTR lpszFileName, UINT nOpenFlags,
                  CFileException* pError = NULL )
```

în care parametri au următoarele semnificaţii:

- `lpszFileName` - este numele fişierului ce urmează a fi deschis, conţinând eventual şi calea;
- `nOpenFlags` este - o mască de biţi care specifică modul de acces la fişier. Variantele posibile pentru acest parametru sunt prezentate în tabelul 1. Este posibilă combinarea mai multor opţiuni, prin intermediul operaţiei logice SAU ("|");

Tabelul 9.1

Valoare	Semnificaţie
<code>CFile::modeCreate</code>	Specifică crearea unui fişier nou. Dacă există deja un fişier cu acelaşi nume, acesta este trunchiat la dimensiune 0;
<code>CFile::modeNoTruncate</code>	Acest parametru, împreună cu cel anterior, va permite păstrarea neschimbată a unui fişierului existent. Este avantajoasă utilizarea lui, pentru că, în acest caz, fişierul va fi sigur deschis, cu conţinutul nul dacă nu exista, sau cu conţinutul curent, în caz contrar;
<code>CFile::modeRead</code>	Fişierul este deschis doar în citire;
<code>CFile::modeReadWrite</code>	Fişierul este deschis în citire şi scriere;
<code>CFile::modeWrite</code>	Fişierul este deschis în scriere
<code>CFile::modeNoInherit</code>	Fişierul nu va putea fi moştenit de procese copil;
<code>CFile::shareDenyNone</code>	Deschide fişierul astfel încât orice proces activ să poată scrie sau citi în el. Crearea fişierului eşuează dacă un alt proces l-a deschis anterior;
<code>CFile::shareDenyRead</code>	Deschide fişierul în aşa fel încât alte procese active nu au drept de citire a datelor din fişier;
<code>CFile::shareDenyWrite</code>	Deschide fişierul în aşa fel încât alte procese active nu au drept de scriere în fişier;
<code>CFile::shareExclusive</code>	Deschide fişierul în aşa fel încât alte procese active nu au drept de scriere sau citire în/din fişier;
<code>CFile::typeText</code>	Fişierul este interpretat ca text, fiind interpretate corespunzător caracterele CR, LF, FF, etc;
<code>CFile::typeBinary</code>	Fişierul conţine date binare

O secvenţă tipică de deschidere în acest mod a unui fişier este prezentată mai jos:

```
CString strFileName="C:\\Users\\Date\\Fisier.dat";
CFileException Fex;
CFile Fisier;
BOOL bResult=Fisier.Open(strFileName, CFile::modeCreate
    | CFile::modeNoTruncate, &Fex);
if (!bResult)
    AfxMessageBox(" Eroare deschidere fisier " + strFileName);
Fex.GetErrorMessage();
```

- `pError` - este un pointer spre un obiect excepție de tip fișier, care va intercepta excepțiile apărute la deschiderea fișierului.

Există o mulțime de cauze de apariție a erorilor la deschiderea unui fișier, cea mai des întâlnită fiind o încercare de deschidere a unui fișier inexistent, sau a unui fișier deschis exclusiv de un alt proces. Funcția `Open()` returnează valoarea `TRUE` dacă deschiderea este reușită, sau `FALSE` în caz contrar, putându-se detecta succesul sau insuccesul operației de deschidere. Din păcate, acest mod de testare a operației, nu dă informații, în caz de eșec, asupra cauzei care a produs eșecul.

Pentru determinarea cauzei, MFC oferă clasa `CFileException`, care încapsulează erorile de lucru cu fișierele. Clasa `CFile` furnizează și un constructor de deschidere a fișierelor, care utilizat, va arunca o excepție `CFileException` dacă deschiderea eșuează. Aceasta poate fi interceptată într-un bloc `try-catch` tipic, ca în exemplul de mai jos:

```
try
{
    CFile Fisier("C:\\Users\\Date\\Fisier.dat",
        ▲ CFile::modeCreate | CFile::modeNoTruncate);
}
catch (CFileException* fEx)
{
    TCHAR buf[255];
    fEx->GetErrorMessage(buf, 255);
    CString strMesErr(buf);
    AfxMessageBox(strMesErr);
}
```

Funcția `Open()` nu aruncă o excepție, dar primește ca parametru un pointer spre un obiect `CExceptionFile`. Dacă deschiderea fișierului eșuează, acest obiect este actualizat cu informații despre tipul erorii. Aceste informații pot fi utilizate, ca în codul de mai jos:

```
CString strFileName="C:\\Users\\Date\\Fisier.dat";
CFile Fisier;
CFileException fEx;
if (!Fisier.Open(strFileName, CFile::modeCreate
    ▲ | CFile::modeNoTruncate, &fEx)
{
    TCHAR buf[255];
    fEx.GetErrorMessage(buf, 255);
    CString strMesErr(buf);
    AfxMessageBox(strMesErr);
}
```

9.2.2 Închiderea fișierelor

Închiderea fișierelor se face prin intermediul funcției `CFile::Close()`. Nu întotdeauna este absolut necesară utilizarea acestei funcții, pentru că destructorul clasei `CFile` apelează automat această funcție când obiectul `CFile` curent iese din domeniul de definiție. Dar, este totuși recomandat ca orice apel al funcției `CFile::Open()` să fie pereche cu un apel al funcției `CFile::Close()`.

Funcția `Close()` poate fi de asemenea utilizată pentru a detașa obiectul **CFile** care o apelează de fișierul pe care acesta îl reprezintă la un moment dat și a-l asocia unui alt fișier. Un exemplu tipic este prezentat mai jos, unde sunt create 3 fișiere cu nume diferite, stocate într-un tablou de nume, utilizând același obiect **CFile**.

```
CString strFilename[3]={"Fisier1.txt", "Fisier2.txt", "Fisier3.txt"};
CFile Fisier;

For (int i=0;i<3;i++)
{
    Fisier.Open(strFilename[i], CFile::modeCreate);
    Fisier.Close();
}
```

9.2.3 Citirea și scrierea din/în fișiere

Citirea și scrierea directă din fișiere se pot face cu funcțiile clasei **CFile** `Read()` și `Write()`, funcții care apelează funcțiile `ReadFile()` și `WriteFile()` ale Windows API. Să ne reamintim că prin intermediul clasei **CFile**, scrierea și citirea se fac direct pe disc, lucru destul de riscant. De aceea, este preferabil să se utilizeze funcții care lucrează cu fluxuri de date I/O. În C++, clasa care implementează aceste funcții este **iostream**, punând la dispoziție funcții cum ar fi `fopen()`, `fseek()`, `fread()`, `fwrite()`, etc.

Într-un proiect MFC apelul acestor funcții este anacronic. În MFC, operațiile cu fișiere în flux, sunt implementate de clasa **CStdioFile**. Funcțiile `Read()` și `Write()` din această clasă, spre deosebire de cele ale clasei **CFile**, oferă acces la fișiere prin intermediul unor fluxuri I/O și nu prin acces direct la disc.

Fișierele asociate unui obiect **CStdioFile** pot fi deschise în mod text sau în mod binar. Fișierele deschise în mod text realizează interpretarea caracterelor de control speciale, cum ar fi CR, LF, etc. Pentru deschiderea unui fișier în mod text, se va utiliza opțiunea `CFile::typeText`.

```
CStdioFile Fisier.Open(Filename, CFile::modeRead | CFile::typeText);
```

Pentru deschiderea fișierului în mod binar, se va utiliza opțiunea `CFile::typeBinary`. În acest mod, toți octeții fișierului sunt interpretați în mod identic, fără a exista coduri de control.

Citirea datelor se poate face prin intermediul a două funcții:

```
virtual UINT Read( void* lpBuf, UINT nCount )
virtual LPTSTR ReadString( LPTSTR lpstr, UINT nMax )
sau
BOOL ReadString(CString& rString)
```

Ambele funcții primesc ca și argumente un pointer la un buffer tampon în care se vor citi datele, respectiv la un șir de caractere, precum și dimensiunea maximă a acestora. În a doua variantă a funcției `ReadString()` se specifică un șir în care se stochează datele citite din fișier. Funcția returnează valoarea `TRUE` dacă citirea e reușită, respectiv `FALSE` dacă citirea eșuează. Pentru toate funcțiile, în caz de eșec se, va arunca o excepție de clasă **CFileException**.

Scrierea datelor se face cu funcțiile:

```
virtual void Write( const void* lpBuf, UINT nCount )
virtual void WriteString( LPCTSTR lpsz )
```

cu aceleași semnificații pentru argumente ca și în cazul anterior. De asemenea, în caz de insucces, este aruncată o excepție **CFileException**.

Pentru exemplificare, vom crea un schelet de program de testare a cunoștințelor. Acest program va prelua întrebările, răspunsurile posibile precum și răspunsul corect, din fișierul *res.dat* aflat în directorul implicit. Acest fișier de tip text, creat cu un editor oarecare, va conține informații cu o structură ce va fi detaliată mai jos. Fiecare linie a fișierului reprezintă informația aferentă unei întrebări.

Fișierul este citit inițial în memorie, pentru a asigura accesul rapid la informație și pentru a evita menținerea pentru un timp îndelungat a fișierului deschis. Testul constă dintr-un număr prestabilit de întrebări. Acestea sunt alese prin generarea câte unui număr aleator mai mic decât numărul întrebărilor memorate. După selectarea unei întrebări, aceasta este eliminată din memorie, pentru a nu putea fi afișată din nou.

Pentru fiecare întrebare sunt afișate trei răspunsuri, doar unul fiind corect. După parcurgerea numărului prestabilit de întrebări, se afișează o notă.

Fie *fis* denumirea proiectului de tip **Dialog Based**. Macheta aplicației este cea de mai jos:

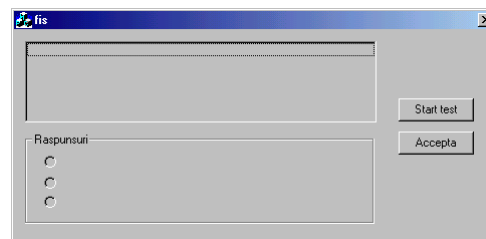


Figura 9.3. Macheta aplicației

Se parcurg pașii:

- se atribuie machetei identificatorul **IDD_FIS_DIALOG**;
- se șterg butoanele **OK** și **Cancel**;
- se atribuie casetei cu listă identificatorul **IDC_TEXT**. Se selectează **single** la **Style-Selection**; De asemenea, în **ClassWizard**, se asociază casetei variabila de categorie **Control**, **CListBox m_lbText**;
- se inserează caseta de grupare **Raspunsuri**, care conține grupul de butoane de opțiune **IDC_RASPUNS1**, **IDC_RASPUNS2**, **IDC_RASPUNS3**. Primul buton va avea opțiunile **Group** și **Tab stop**, iar celelalte două doar **Tab stop**; În **ClassWizard**, se asociază grupului de butoane, variabila de categorie **Value**, **int m_nRaspuns**; Răspunsurile posibile vor fi afișate ca și texte asociate butoanelor;
- se inserează butoanele **Start test** și **Acceptă**, cu identificatorii **IDC_START_TEST** și respectiv **IDC_ACCEPTA**;
- deoarece caseta cu listă nu interpretează caracterele de control, în fișierul cu date vom insera marcatori pentru delimitarea diferitelor componente ale întrebărilor. Spre exemplu, fișierul de test *res.dat* va avea conținutul:

Aceasta este prima întrebare. Urmează să vad cum se @asează în caseta și să generez răspunsuri@~Acesta este primul răspuns la prima întrebare^Acesta este al doilea răspuns la prima întrebare^Acesta este al treilea răspuns la prima întrebare^|2

Aceasta este a doua întrebare. Urmează să vad cum se @asează în caseta și să generez răspunsuri@~Acesta este primul răspuns la a doua întrebare^Acesta este al doilea răspuns la a doua întrebare^Acesta este al treilea răspuns la a doua întrebare^|2

Aceasta este a treia întrebare. Urmează să vad cum se @asează în caseta și să generez răspunsuri@~Acesta este primul răspuns la a treia întrebare^Acesta este al doilea răspuns de a treia întrebare^Acesta este al treilea răspuns la a treia întrebare^|2

Caracterul “@” este utilizat ca delimitator de linie. La întâlnirea lui, în caseta cu listă se va trece la linie nouă. Sfârșitul textului afișat în caseta cu listă este marcat de delimitatorul “~”. Apoi, urmează cele trei răspunsuri posibile, delimitate prim “^”. Delimitatorul “|” specifică varianta corectă de răspuns.

- se inserează o machetă de tip dialog (**Resource View**, click dreapta, **I**nsert, **D**ialog, **N**ew, etc...), ca în figura 9.4. Se inserează în machetă butoanele de opțiune **Da** și **Nu**, având opțiunea **Push-like** de la **Styles** selectată, cu identificadorii **IDC_DA** și **IDC_NU**, grupate într-o caseta de grupare. **IDC_DA** va avea setate opțiunile **Tab stop** și **Group**, iar **IDC_NU** doar **Tab stop**. De asemenea în **ClassWizard** se asociază grupului de butoane variabila de categorie **value int m_nDa**. Această machetă va fi lansată modal, pentru a relua testul după terminarea lui, sau a renunța la programul de testare. Fie **CModalDlg** clasa care mapează această casetă.

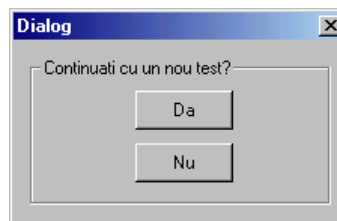


Figura 9.4. Macheta de reluare

- se modifică funcția `OnInitDialog()`, astfel încât dacă fișierul de întrebări este deschis corect testarea să fie posibilă, altfel butoanele **Start test** și **Acceptă** să fie inactice:

```

BOOL CFisDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
    ...
    // TODO: Add extra initialization here
    GetDlgItem(IDC_ACCEPTA)->EnableWindow(FALSE);
    if (!CompleteazaSetIntrebări())
        GetDlgItem(IDC_START_TEST)->EnableWindow(FALSE);

    return TRUE; // return TRUE unless you set ...
}

```

- funcția `CompleteazăSetIntrebări()` returnează valoarea **TRUE** dacă fișierul de întrebări a fost deschis corect și **FALSE** în caz contrar. Ea se inserează în clasa **CFisDlg** ca funcție membru (**Add Member Function**) și are implementarea de mai jos.

```

int CFisDlg::CompleteazaSetIntrebari()
{
    CString strFilename("c:\\Html\\res.dat");
    TCHAR cBuf[255];

    CStdioFile dataFile;
    CFileException fEx;

    if (!dataFile.Open(strFilename, CFile::modeRead
        ▲ | CFile::typeText, &fEx))
    {
        TCHAR cBufEx[255];
        fEx.GetErrorMessage(cBufEx, 255);
        CString strMesErr(cBufEx);
        AfxMessageBox(strMesErr);
        return 0;
    }
    else
    {
        cstrDepozit.RemoveAll();
        pos=cstrDepozit.GetHeadPosition();
        while (dataFile.ReadString(cBuf,255))
        {
            CString strBuf(cBuf);
            cstrDepozit.AddTail(strBuf);
        }
        dataFile.Close();
        return 1;
    }
}

```

Funcția deschide fișierul de test în mod text, în citire. Dacă deschiderea eșuează, este aruncată o excepție **CFileException** care este interceptată și se afișează mesajul de eroare corespunzător. În acest caz, funcția returnează valoarea 0.

Dacă deschiderea este reușită, se șterge conținutul variabilei **CStringList** **cstrDepozit**, în care va fi încărcat conținutul fișierului. Apoi, atâta timp cât se pot citi date din fișier, conținutul bufferului în care se citesc datele va fi adăugat la coada variabilei **cstrDepozit**. În final, fișierul este închis, iar funcția returnează valoarea 1. Variabila **cstrDepozit** este adăugată clasei **CFisDlg** ca variabilă membru (**Add Member Variable, ...**);

De asemenea, se inserează variabila **POSITION** **pos** ca membră a clasei **CFisDlg**.

- testul este lansat la apăsarea butonului **Start test**. Funcția care implementează apăsarea acestui buton este

```

void CFisDlg::OnStartTest()
{
    // TODO: Add your control notification handler code here
    Nota=0;
    m_nRaspuns=0;
    UpdateData(FALSE);
    NumarIntrebariInTest=2;
    if (cstrDepozit.GetCount())
    {
        GetDlgItem(IDC_ACCEPTA)->EnableWindow(TRUE);
    }
}

```

```

        GetDlgItem(IDC_START_TEST)->EnableWindow(FALSE);
        GenereazaTest();
    }
    else
    {
        AfxMessageBox(" Nu mai exista intrebari! ");
        CDialog::OnOK();
    }
}

```

Variabilele `int Nota` și `int NumarIntrebariInTest` sunt adăugate clasei `CFisDlg` ca variabile membru. Ele vor memora nota obținută și numărul de întrebări în care constă textul. Testul poate fi continuat atâta timp cât mai există întrebări în memorie, altfel se afișează un mesaj de terminare și se apelează funcția `OnOK()` pentru a termina programul.

- generarea testului se face cu funcția `GenereazaTest()`. Aceasta este adăugată clasei `CFisDlg` ca și funcție membru și are implementarea:

```

void CFisDlg::GenereazaTest()
{
    CString strAfiseaza;
    CString strTemp;

    srand((unsigned) time (NULL));
    int nNumarIntrebare=rand() % (cstrDepozit.GetCount());

    CautaPozitia(nNumarIntrebare);
    strAfiseaza=cstrDepozit.GetAt(pos);

    int i=0;

    m_lbText.ResetContent();
    while (strAfiseaza[i] != '~')
    {
        strTemp.Empty();
        for (i=i;strAfiseaza[i] != '@';i++)
            strTemp+=strAfiseaza[i];
        m_lbText.AddString(strTemp);
        i++;
    }
    i++;
    int j=0;
    while (strAfiseaza[i] != '|')
    {
        strTemp.Empty();
        for (i=i;strAfiseaza[i] != '^';i++)
            strTemp+=strAfiseaza[i];

        GetDlgItem(IDC_RASPUNS1+j)->SetWindowText(strTemp);
        j++;
        i++;
    }
    i++;
    RaspunsCorect=strAfiseaza[i]-'1';
    cstrDepozit.RemoveAt(pos);
}

```

Se generează un număr aleator, care va reprezenta numărul întrebării selectate. Apoi, se parcurge lista `strDepozit` până la înregistrarea selectată, prin intermediul funcției `CautaPozitia()`. Înregistrarea este stocată în șirul `strAfiseaza`. Urmează interpretarea informației din această variabilă, afișarea textului corespunzător în caseta cu listă și afișarea etichetelor butoanelor de opțiune. În final, înregistrarea selectată este eliminată din `strDepozit`, pentru ca aceeași întrebare să nu poată fi selectată din nou, respectiv pentru a se genera un număr aleator într-o plajă de valori cu maximumul decrementat cu 1.

Variabila `int RaspunsCorect` este adăugată ca și variabilă membru a clasei `CFisDlg`.

- funcția `CautaPozitia()` este adăugată clasei `CFileDlg` ca și funcție membru și are implementarea:

```
void CFisDlg::CautaPozitia(int pozitia)
{
    pos=cstrDepozit.GetHeadPosition();
    for (int i=0;i<pozitia;i++)
        cstrDepozit.GetNext(pos);
}
```

- după selectarea unui răspuns, acesta este validat prin apăsarea butonului **Accepta**. Implementarea funcției care răspunde la apăsarea acestui buton este:

```
void CFisDlg::OnAccepta()
{
    // TODO: Add your control notification handler code here
    UpdateData();
    NumarIntrebariInTest--;

    if (m_nRaspuns==RaspunsCorect) Nota++;
    if (NumarIntrebariInTest) {
        if (cstrDepozit.GetCount()) GenereazaTest();
        else {
            AfxMessageBox(" Nu mai exista intrebari! ");
            CDialog::OnOK();
        }
    }
    else
    {
        CString strNota;
        CString strTemp;

        strTemp.Format(" %d ",Nota);
        strNota+="La acest test ati obtinut nota "+strTemp;
        MessageBox(strNota,"REZULTAT",MB_ICONSTOP);
        CRaspunsDlg dlgCasetaRaspuns(this);
        int nRetCode=dlgCasetaRaspuns.DoModal();
        if (!dlgCasetaRaspuns.m_nDa)
        {
            CompleteazaSetIntrebari();
            Nota=0;
            m_nRaspuns=0;
            UpdateData(FALSE);
            NumarIntrebariInTest=2;
            if (cstrDepozit.GetCount()) GenereazaTest();
            else

```

```

        {
            AfxMessageBox(" Nu mai exista intrebari! ");
            CDialog::OnOK();
        }
    }
    else CDialog::OnOK();
}
}

```

Funcția decrementează numărul de întrebări rămase, iar dacă răspunsul a fost corect, incrementează nota. Dacă mai există întrebări în test și mai există întrebări disponibile, generează o nouă întrebare, altfel termină programul.

Dacă nu mai există întrebări în test, se afișează nota și se lansează caseta modală pentru a se vedea dacă se începe un nou test sau se termină programul. Dacă se dorește reluarea testului și mai există întrebări disponibile se reia testul, altfel se iese.

- se generează funcțiile `OnDa()` și `OnNu()` care răspund mesajului `BN_CLICKED` asociat butoanelor casetei derivate:

```

void C RaspunsDlg::OnDa()
{
    // TODO: Add your control notification handler code here
    UpdateData();
    CDialog::OnOK();
}

void C RaspunsDlg::OnNu()
{
    // TODO: Add your control notification handler code here
    UpdateData();
    CDialog::OnOK();
}

```

Funcțiile vor actualiza valoarea `m_nDa`, testată în macheta părinte, după care vor închide caseta modală.

- pentru a putea lansa modal macheta, va trebui ca la începutul fișierului *fisDlg.cpp* să se insereze linia

```

...
#include "stdafx.h"
#include "fis.h"
#include "fisDlg.h"
#include "ModalDlg.h"

#ifdef _DEBUG
...

```

În acest moment programul este complet funcțional.

Uzual, datele conținute în fișierul de teste trebuie codificate, în caz contrar putând fi citite cu orice editor de texte. Vom crea un nou program, care, presupune că fișierul cu teste a fost deja creat cu un editor oarecare, fiind salvat pe disc cu un nume oarecare. Programul va primi ca intrare numele fișierului, iar la apăsarea butonului **Codifica** va executa următoarele:

- va deschide în citire fișierul cu teste;
- dacă deschiderea este reușită, va crea fișierul *res.dat*;
- va citi linie cu linie fișierul cu teste, adunând 1 la fiecare cod ASCII impar și scăzând 1 din fiecare cod ASCII par;
- va scrie fiecare linie astfel codificată în fișierul *res.dat*;
- dacă o operație de deschidere eșuează, va afișa un mesaj de eroare și va relua dialogul;

Pentru aceasta se parcurg următorii pași:

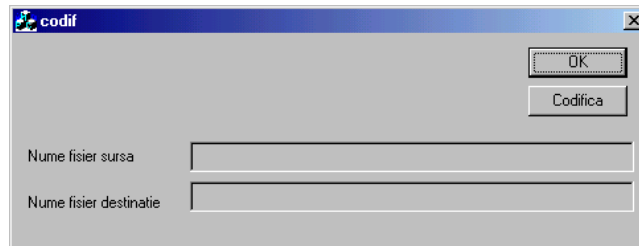


Figura 9.5. Macheta programului

- se deschide un proiect numit *codif*, realizându-se pentru acesta macheta din fig. 9.5;
- se modifică identificatorul casetei de editare de sus în `IDC_NUMEFS`, iar în **ClassWizard** se asociază acestei casete variabila de categorie **Value** `CString m_strNumefs`;
- se modifică identificatorul casetei de editare de jos în `IDC_NUMEFD`, iar în **ClassWizard** se asociază acestei casete variabila de categorie **Value** `CString m_strNumefd`;
- se modifică identificatorul butonului **Codifica** în `IDC_CODIF`;
- se asociază mesajului `BN_CLICKED` generat de butonul **Codifica** funcția `OnCodif()` cu implementarea de mai jos:

```
void CCodifDlg::OnCodif()
{
    // TODO: Add your control notification handler code here

    UpdateData();
    if (m_strNumefd.IsEmpty())
        m_strNumefd="res.cod";
    if (!m_strNumefs.IsEmpty())
    {
        TCHAR cBuf[255];
        CStdioFile outputFile;

        try
        {
            CStdioFile inputFile(m_strNumefs,CFile::modeRead
                ▲ |CFile::typeText);
            BOOL bResult=outputFile.Open(m_strNumefd,
                ▲
                CFile::modeCreate|CFile::modeWrite|CFile::typeText);
            if (!bResult)
                AfxMessageBox("Eroare creare fisier "+m_strNumefd);
            else
            {
                while (inputFile.ReadString(cBuf,255))
```



```

        {
            CString strBuf(cBuf);
            CString outBuf;
            outBuf.Empty();
            for (int i=0;i<strBuf.GetLength();i++)
            {
                char x;
                if (i%2) x=strBuf[i]+1;
                else x=strBuf[i]-1;
                outBuf+=x;
            }
            outputFile.WriteString(outBuf);
        }
        outputFile.Close();
        AfxMessageBox("Fișier codificat ! Datele se afla in
            ▲ fișierul "+m_strNumefd);
    }
    inputFile.Close();
}
catch (CFileException* fEx)
{
    TCHAR cBufEx[255];
    fEx->GetErrorMessage(cBufEx, 255);
    AfxMessageBox(cBufEx);
}
}
}

```

Observație: În cele 2 programe de prezentate anterior au fost folosite în scop didactic cele trei metode de detectare a erorilor apărute în manipularea fișierelor. În mod normal, într-un program se utilizează în mod consecvent doar una din ele.

Observație: Fișierele text sunt citite și scrise în mod secvențial. Pentru fișierele cu conținut binar, poate fi uneori nevoie de acces direct la înregistrări. Orice fișier deschis menține un pointer la înregistrarea curentă care este accesată. Acest pointer poate fi poziționat cu funcția

virtual LONG Seek(LONG lOff, UINT nFrom)

unde lOff reprezintă numărul de octeți peste care se mută pointerul, iar nFrom modalitatea în care acesta se numără. Parametrul nFrom ia valorile din tabelul de mai jos:

Tabelul 9.2

Valoare	Semnificație
CFile::begin	Mută pointerul lOff octeți înainte, numărați de la începutul fișierului; lOff trebuie să fie pozitiv;
CFile::current	Mută pointerul lOff octeți, numărați de la poziția curentă; mutarea se face înainte dacă lOff>0, respectiv înapoi dacă lOff<0;
CFile::end	Mută pointerul lOff octeți înapoi, numărați de la sfârșitul fișierului; lOff trebuie să fie negativ;

Întrebări și probleme propuse

1. Implementați și executați toate exemplele propuse în capitolul 9;

2. Modificați programul de la pagina 179 ca mai jos:

```
...
void Test::ThrowString()
{
    // throw "EXCEPTIE TIP SIR DE CARACTERE!!";
    throw int(4);
}
...
// catch(...) {
//     ;
// }
```

...
Ce se întâmpla la execuție? Explicați.

3. Modificați din nou programul ca mai jos:

```
...
catch(...) {
    cout << "ALT FEL DE EXCEPTIE!\n";
}
...
```

...
Ce se întâmpla la execuție? Explicați.

4. Modificați în același program primul bloc `catch` ca mai jos:

```
...
// catch(char* sir1_Exception){
//     cout<< sir1_Exception<< "\n";
//     catch (int exceptie_intreaga) {
//         cout << "EXCEPTIE DE VALOARE " << exceptie_intreaga << "\n";
//     }
// }
```

...
Ce se întâmpla la execuție? Explicați.

5. Modificați programul de testare, astfel încât să lucreze cu fișierul codificat (implementați o fază de decodificare a fișierului înainte de salvarea acestuia în memorie);
6. Scrieți un program de concatenare a două fișiere de tip text;