Міністерство освіти і науки України
Уманський державний педагогічний університет імені Павла Тичини

Мова програмування Lazarus

Навчальний посібник
Частина 3

Укладач Паршукова Л. М.

Умань
2021

УДК 004(075.8)

Мова програмування Lazarus: навчальний посібник. Частина 3 / Уманський держ. пед. ун-т імені Павла Тичини ; уклад. Паршукова Л.М. – Умань : Візаві, 2021. - 139 с.

Навчальний посібник «Мова програмування Lazarus» висвітлює основні теми розділів з основ подійно- та об’єктно-орієнтованого програмування. В якості середовища програмування розглядається безкоштовне середовище програмування Lazarus на мові програмування Object Pascal (безкоштовний аналог Delphi)

УДК 004(075.8)
© Л.М. Паршукова, уклад., 2021

Зміст

1.	Тема 21. Консольні додатки й параметри програми	4
2.	Тема 22. Тип TStringList. Робота з папками	14
3.	Тема 23. Файли	21
4.	Тема 24. Записи й варіант. Сітка рядків TStringGrid	38
5.	Тема 25. Телефонний довідник (проект)	54
6.	Тема 26. DLL	70
7.	Тема 27. Тестування й налагодження	95
8.	Тема 28. Створення довідкової системи	105
9.	Тема 29. Створення інсталятора	124
10.	Список використаних джерел та корисні посилання	139

Тема 21. Консольні додатки й параметри програми

1. Консольні додатки
2. Створення консольного додатка
3. WRITE і WRITELN

1. Консольні додатки

На сьогоднішній день про програмування на Lazarus існує дуже мало книг російською мовою, і всі вони описують роботу в основному, з консольними додатками, хоча особливої потреби в консолях зараз немає. Але й зовсім обійти цю тему не можна. Що ж такий консольний додаток?

Консольним додатком називається програма, що не має графічного інтерфейсу - вікон, і яка працює в текстовому режими в чорно-білій консолі. Команди в такій програми доводитися вводити із клавіатури, результати роботи консольні додатки також виводять на екран у текстовому режими.

Якщо ви користуєтеся операційною системою Windows, то натисніть на «**Пуск**» і виберіть команду «**Виконати**». У вікні, що відкрилося, наберіть

cmd

і натисніть **<Enter>**. Відкриється командна консоль, у якій команди потрібно вводити в текстовому режими:

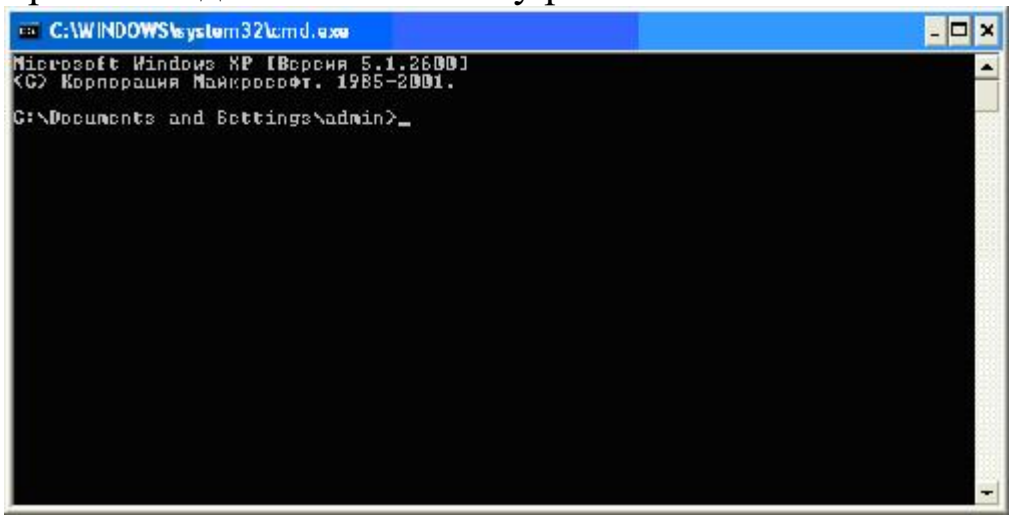


Рис. 21.1. Консоль cmd.exe

Властиво, ви бачите вікно предка Windows - операційної системи MS-DOS. Саме так виглядав екран із завантаженою ОС MS-DOS, і управляти нею доводилося, вручну набиваючи всілякі команди. Це вже

пізніше корпорація Microsoft навісила на ядро MS-DOS усілякі драйвери й утиліти, постачила його графічним віконним інтерфейсом, і з'явився спочатку напівграфічна ОС Windows 3.10 (росіянин варіант був 3.11 версії), а потім і цілком графічна Windows 95. Але до консольного ядра системи можна звертатися й у сучасних ОС, в Windows для цього використається програма cmd.exe, а в ОС Linux - терминал.

Всі мови високого рівня дозволяють робити й консольні додатки, інша справа - навіщо? Переважна більшість сучасних програм має графічний інтерфейс, що ми з вами створювали з найпершої лекції. А консольні додатки робляться в основному, системними програмістами. Незважаючи на те, що Object Pascal володіє для цього всіма необхідними інструментами, системщики звичайно користуються такими мовами, як Асемблер, C, рідше - C++. Але все-таки знати, як створюються консольні додатки потрібно, тому дану тему ми присвячуємо їм.

2. Створення консольного додатка

Створити консольний додаток можна різними способами, але простіше всього так. Відкрийте **Lazarus**. Командою «**Проєкт -> Закрити проєкт**» закрийте поточний проєкт, автоматично з'явиться вікно **Майстра створення проєкту**. У ньому натиснемо кнопку «**Новий проєкт**». З'явиться вікно створення проєкту, у якому можна вибрати «**Програма**» або «**Консольний додаток**». Якщо ми виберемо «**Програма**», то буде створений модуль із мінімальним кодом. Якщо ж ми виберемо «**Консольний додаток**», то коду буде більше, тому що при цьому створюється програма з новим класом, похідним від TCustomApplication. TCustomApplication забезпечує гарну основу й робить програмування утиліт командного рядка досить простим. Наприклад, перевірку опцій командного рядка, написання довідки, перевірку змінні оточення й обробку виключень. Всі програми LCL автоматично це використають.

Але нам не потрібно використати можливості класу TCustomApplication, ми робимо просту консольну програму, тому у вікні створення проєкту ми вибираємо «**Програма**». Сформується проєкт, а в **Редакторі вихідного коду** буде минімум тексту:

```
program Project1;

{$mode objfpc}{$H+}
```

```
uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

begin
end.
```

Зверніть увагу, якщо нам потрібно підключити до програми які те модулі, то робити це потрібно до коментарю

```
  { you can add units after this };
```

У цьому випадку ми ставимо кому після зазначеного модуля Classes і додаємо свої модулі. Завершальна крапка з коми коштує після коментарю, нам неї ставити не потрібно.

Свій код ми будемо писати миж дужками

```
begin
end.
```

А якщо нам буде потрібно вказати глобальні константи, змінні або зробити оголошення процедур і функцій, то все це робиться до begin.

Для приклада ми створимо простий додаток, а по ходу справи, познайомимося з інструментами введення-виведення інформації в консольних додатках.

3. WRITE і WRITELN

Процедура Write призначена для виведення інформації на екран. Вона має наступний синтаксис:

```
procedure Write(var F:Text; Args:Arguments);
```

Діє процедура в такий спосіб. У дужках ми можемо вказати якийсь текст, вивести вміст змінних. Наприклад:

```
  write('Усім привіт!!!');
  write('Змінна a = ', a, '; змінна b = ', b);
```

Після виведення інформації на екран курсор залишається на тім же рядку, у позиції, де він виявився після виведення останнього символу.

Процедура Writeln діє точно та само, але після виведення останнього символу курсор переходить на початок наступного рядка.

Для ознайомлення із процедурами створіть новий проєкт «**Програма**». Модуль можна не перейменовувати, просто збережіть його в папку **21-01**. Повний текст модуля наступний:

```
program project1;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

var a, b: Integer;
begin
  a:= 5;
  b:= 10;
  writeln('Усім привіт!!!');
  writeln('Змінна a = ', a, '; змінна b = ', b);
  readln();
end.
```

Зверніть увагу: розділ змінних var ми вказали до службового слова begin, тобто, змінні a і b у межах модуля є глобальними. Далі ми привласнили змінним значення, потім вивели на екран вітання, а потім вміст змінних. На процедуру readln() поки не звертайте уваги, вона потрібна тільки, щоб програма не закрилася відразу ж після виведення тексту, а була на екрані, поки ми не натиснемо **<Enter>**. Збережіть проєкт і запустіть його на виконання. І відразу ж ми бачимо недолік: замість українськиї літер виходить щось, що в Інтернеті називають «кракозябрами»:

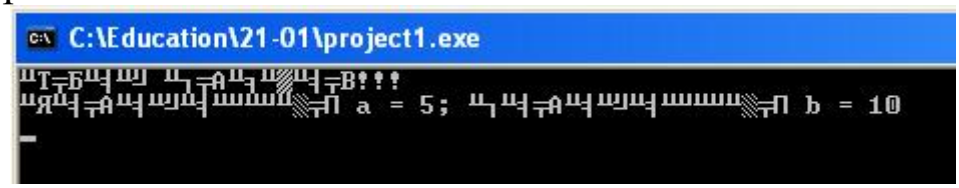


Рис. 21.2. Неправильний висновок кирилиці в консольній програмі

У темі №5 ми згадували, що в консольних додатках Windows прийнято використати кодування CP866, тоді як у графічних додатках використовується CP1251 і Юнікод. Звідси й «кракозябрики». В Linux і Unix таких проблем немає. Однак вирішити цю проблему зовсім нескладно. Натисніть **<Enter>**, щоб закрити консоль, і поверніться в **Редактор коду**. Клацніть по вікну з кодом правою кнопкою миші, і в контекстному меню, що відкрився, виберіть команду **«Параметри файлу -> Кодування -> CP**866». Відкриється вікно зміни кодування, нам треба натиснути кнопку «**Змінити файл**». Знову запустіть програму на виконання - тепер з кодуванням повний порядок:

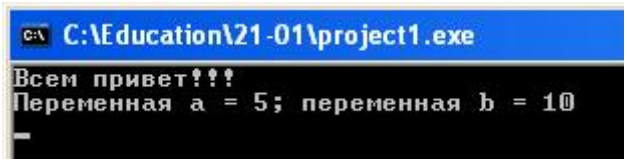


Рис. 21.3. Правильний висновок кирилиці в консольній програмі

READ і READLN

Процедура Read призначена для введення інформації в програму, точніше, у змінні. Процедура працює в такий спосіб:

read(Змінна1, Змінна2, ..., Змінна);

Тобто, як параметри ми вказуємо одну або розділені комами змінні. Коли програма доходить до цього місця, то очікує від користувача введення даних. Користувач повинен ввести дані такого ж типу, як у відповідних змінних. Якщо змінних не багато, то відповідні значення потрібно вводити, розділяючи їх пробілами. Наприклад, якщо є змінна цілого типу i, і дійсна змінна r, то при виконанні інструкції

read(i, r);

відбудеться от що: програма призупинить свою роботу, очікуючи від користувача введення даних. Користувач повинен ввести ціле й дійсне числа, розділені пробілом, і нажати **<Enter>**. Наприклад, користувач ввів:

12 3.45

Як тільки він натисне **<Enter>**, у змінну i потрапить число 12, а в r - число 3,45. Якщо користувач введе некоректні дані, що не відповідають типу змінної, то виникає помилка й програма аварійно завершує роботу.

Користувач може ввести більше даних, чим зазначено змінних у процедурі read. У цьому випадку, дані що залишилися будуть оброблені наступною процедурою read. Наприклад, користувач ввів:

1 23 456

і нажав **<Enter>**. Ці дані можуть бути оброблені наступними інструкціями:

```
read(a, b);
read(c);
```

Процедура Readln працює також. Відмінність полягає в тім, що якщо користувач ввів більше даних, то readln не передає зайві дані в наступну інструкцію readln, а просто обрубує їх. Тобто, якщо змінити попередній приклад:

```
readln(a, b);
read(c);
```

то в a і b потраплять числа 1 і 23, після чого програма буде очікувати наступних даних, а число 456 втратиться.

Якщо вказати read або readln без параметрів, наприклад, так:

```
readln();
```

то програма просто перейде в режим очікування, поки користувач не введе **<Enter>**. Наприкінці консольних додатків завжди бажано вказувати таку порожню readln, інакше програма просто мигне й виключиться, не давши користувачеві побачити результати роботи. Або можна якось інакше забезпечити роботу програми доти, поки користувач сам не захоче її закрити. Наприклад, іноді програмісти виводять повідомлення типу «Продовжити роботу програми? 0-ні, 1-да». Якщо користувач введе 0, то програма завершиться, а якщо 1, то продовжить роботу. Який спосіб використати - справа користувача.

Щоб користувач розумів, чого від нього хоче програма, перед кожною інструкцією read або readln треба за допомогою write або writeln виводити пояснення.

Повернемося до нашої програми, і небагато змінимо код (між begin і end.):

```
begin
  write('Введіть перше ціле число: ');
  readln(a);
```

```
  write('Введіть друге ціле число: ');
  readln(b);
  writeln('Результат:');
  writeln('Змінна a = ', a, '; змінна b = ', b);
  readln();
end.
```

Тепер програма виводить запит «Введіть перше число:», і завмирає, очікуючи від користувача уведення даних. Після того, як він уведе число й натисне **<Enter>**, програма в такий же спосіб запросить друге число. Потім вона виведе результат, і буде очікувати, поки користувач не натисне **<Enter>**:

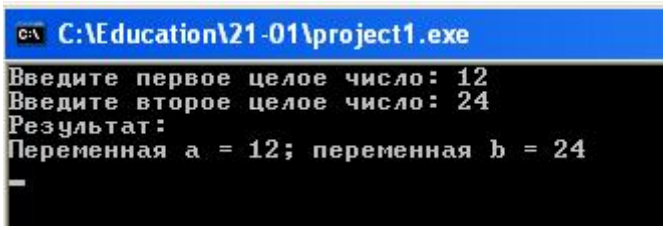


Рис. 21.4. Результат роботи програми

Параметри консольного додатка

Нерідко системні консольні програми викликаються з якимись параметрами. Наприклад, в Windows можна нажати «**Пуск -> Виконати**» і вказати наступну команду:

```
ping 127.0.0.1 -t
```

У результаті буде запущена консольна утиліта ping.exe, що робить трасування (обмін даними) із зазначеною IP-адресою. Таким чином, перевіряють, чи є зв'язок з даною адресою. У прикладі програма запускається із двома параметрами. Першим параметром є IP-адреса 127.0.0.1 - це адреса локальної машини, звертаючись до 127.0.0.1, ви звертаєтеся до свого ж комп'ютера. Другим параметром є

```
-t
```

Цей параметр вказує утиліті, що не слід завершувати обмін даними, поки користувач сам не перерве роботу програми. Якщо не вказати -t, програма зробить 4 спроби відправити дані, після чого завершить свою роботу. З параметром -t програма буде працювати, поки користувач не натисне **<Ctrl+C>**.

Можливо, коли-небудь і нам буде потрібно створити програму (не обов'язково консольну), що вміє приймати й обробляти параметри. Для обробки параметрів використаються дві функції: ParamCount і ParamStr.

ParamCount повертає кількість параметрів, переданих у програму.

ParamStr(i) повертає параметр під індексом і - індексація починається з 1. Наприклад, вивести на екран перший і останній параметри можна так:

```
ShowMessage('Перший параметр: ' + ParamStr(1);
ShowMessage(Останній параметр: ' + ParamStr(ParamCount);
```

Однак майте на увазі, що в будь-якій програми, консольної або звичайної, навіть запущеної зовсім без усяких параметрів, доступний і «нульовий» параметр ParamStr(0) - у ньому завжди перебувають адреса й ім'я файлу програми.

Щоб закріпити матеріал, давайте знову змінимо нашу програму, одержуючи й обробляючи параметри. Ми зробимо так, що її можна буде використати без параметрів, тоді користувач буде вводити числа прямо в програми. Можна буде завантажити програму з одним параметром, тоді це значення потрапить у змінну a, а число для b користувачеві потрібно вводити вручну. І, нарешті, програму можна буде завантажувати з безліччю параметрів, при цьому перші два потраплять у a і b, інші будуть зігноровані. Не забувайте, що тип даних повинен відповідати змінним, тобто один параметр, або перших два параметри повинні бути обов'язково цілими числами. Інакше програма навіть не запуститься, а просто виведе повідомлення про помилку.

Майте на увазі, що всі параметри, навіть числа, вводяться у вигляді тексту, так що для роботи із цілими числами нам доведеться використати функції IntToStr і StrToInt. А обидві ці функції описані в модулі SYSUTILS, що нам доведеться підключити в розділі uses. От повний лістинг програми:

```
program project1;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
```

```
  Classes, SysUtils
  { you can add units after this };

var a, b: integer;
begin
  //якщо параметрів взагалі немає:
  if ParamCount = 0 then begin
    write('Введіть перше ціле число: ');
    readln(a);
    write('Введіть друге ціле число: ');
    readln(b);
  end
  //якщо тільки один параметр:
  else if ParamCount = 1 then begin
    a:= StrToInt(ParamStr(1));
    write('Введіть ціле число: ');
    readln(b);
  end
  //якщо параметрів більше одного:
  else begin
    a:= StrToInt(ParamStr(1));
    b:= StrToInt(ParamStr(2));
  end;
  //тепер висновок результатів:
  writeln('Всього параметрів: ' + IntToStr(ParamCount));
  writeln('Запущена програма: ' + ParamStr(0));
  writeln('Змінна a = ', a, '; змінна b = ', b);
  readln();
end.
```

Зверніть увагу, після модуля Classes ми вказали модуль SysUtils. Далі, ми забезпечили введення даних у a і b у випадках, якщо параметрів взагалі немає, якщо є тільки один параметр, і якщо параметрів більше, ніж один. Наприкінці ми виводимо загальну кількість зазначених параметрів, нульовий параметр із зазначенням адреси й файлу програми, і потім вказуємо значення змінних. На завершення вказуємо порожній readln, щоб програма не закрилася, поки не захоче користувач.

Збережіть проєкт і запустіть його. Він завантажиться без параметрів, вам доведеться ввести обидва числа вручну. Введіть числа, помилуйтеся на результат і закрийте програму. Тепер завантажимо її з параметрами. Для цього натисніть «**Пуск -> Виконати**», натисніть кнопку «**Огляд**», знайдіть і відкрийте наш проєкт, допишіть після його імені один або два параметри, наприклад, так:

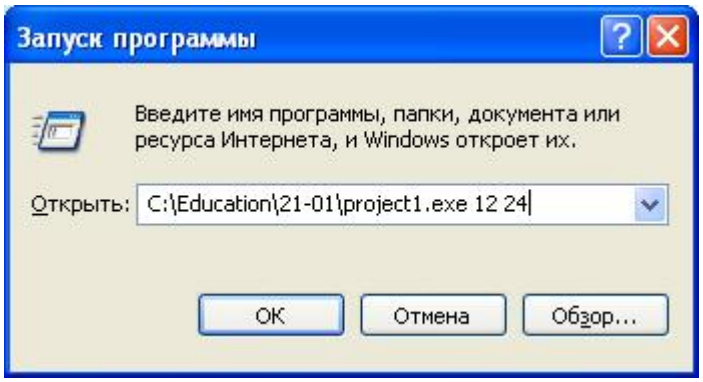


Рис. 21.5. Вказівка параметрів програми

Як бачите, як параметри вказано два числа: 12 і 24. Програма не стане запитувати в мене дані, а відразу виведе результат:

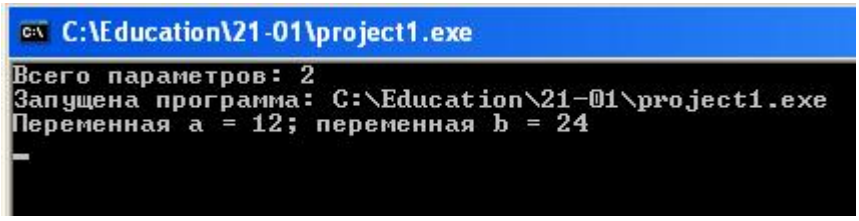


Рис. 21.6. Результат роботи програми

Поекспериментируйте із програмою, викликаючи її з різною кількістю параметрів.

Як уже говорилося, функції ParamCount і ParamStr можна використати в будь-якій програмі, не обов’язково консольної.

Тема 22. Тип TStringList. Робота з папками

1. TStringList
2. Робота з папками
3. Робота з рядками

1.TStringList

Дотепер ми з вами працювали з файлами тільки за допомогою компонентів, наприклад, TMemo. Однак для цього є й інші способи, і найпростіший - тип даних TStringList. TStringList є нащадком класу TStrings, що нам уже зустрічався - властивість Lines в TMemo, і властивість Items у компонентів TListBox і TComboBox. Прямо працювати з типом TStrings не можна, а от з TStringList можна.

TStringList дозволяє створювати текстові файли, записувати в них текст або навпаки, зчитувати його з файлу, додавати рядка й робити з текстом масу інших корисних дій. У змінної типу TStringList дані зберігаються у вигляді окремих рядків, і можна звернутися або до всього тексту цілком, або до однієї з його рядків. Щоб використати цей тип даних, мало оголосити змінну типу TStringList, потрібно ще цю змінну проініціалізувати (створити), а після роботи з нею - знищити. Робиться це в такий спосіб:

```
var
  sl: TStringList;  //оголосили змінну
begin
  sl:= TStringList.Create;  //проініціалізували
  … ;  //якісь дії зі змінної
  sl.Free;  //знищили
```

Коли ми тільки оголосили змінну, компілятор ще не виділив під неї пам’ять. А от коли ми викликали метод Create, те в ОЗУ для змінної sl була виділена пам’ять. Сама змінна типу TStringList є покажчиком - вона вказує на текст, що зберігається в пам’яті, а також кількість займаних текстом байт. Після ініціалізації змінної можна користуватися: створювати з її допомогою файли, обробляти текст і так далі. А от коли ми викликаємо метод Free, ми звільняємо пам’ять, займану змінною, знищуємо її. Після цього зі змінною працювати вже не можна, спроба звертання до неї викличе помилку.

Спробуємо використати цей тип на практиці. Відкрийте **Lazarus** з новим проєктом. Відразу ж збережіть його в папку **22-01**, перейменовувати форму й проєкт не будемо, залишимо імена за замовчуванням.

У властивості Caption форми напишіть Використання TStringList. Додайте на форму один компонентів TEdit і чотири простих кнопки **TButton**. В TEdit очистіть властивість Text, а у властивостях Caption кнопок напишіть, відповідно:

Додати рядок у текст
Вивести текст на екран
Зберегти у файл
Читати з файлу

У результаті у вас повинна вийти приблизно така форма:

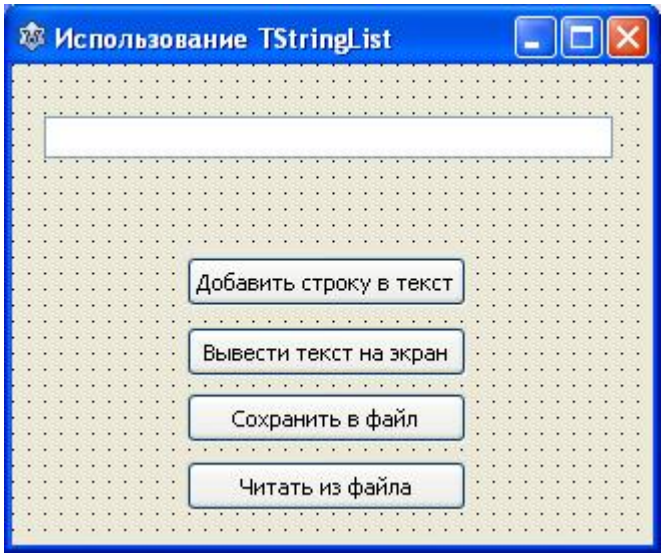


Рис. 22.1. Форма програми

Щоб зі змінною можна було працювати звідусіль, оголосимо її, як глобальну, над службовим словом implementation:

```
var
  Form1: TForm1;
  sl: TStringList;

implementation

{$R *.lfm}
```

Рис. 22.2. Оголошення глобальної змінної

Оскільки змінна глобальна, то виділяти під неї пам’ять потрібно відразу, до натискань на кнопки. У подібних випадках для підготовчих операцій, таких як ініціалізація глобальних змінних, наприклад, найкраще підходить подія форми OnCreate. А для звільнення пам’яті перед закриттям програми найкраще використати властивість форми

OnClose. Виділіть форму, перейдіть на вкладку **Події Інспектори об'єктів**, і двічі клацніть по події OnCreate, генеруючи код для неї. Код події дуже простий:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  //ініціалізуємо змінну:
  sl:= TStringList.Create;
end;
```

Тут ми просто виділили пам'ять для глобальної змінної, і зробили це відразу на початку роботи програми, ще до показу форми на екрані. Код події OnClose не складніший:

```
procedure TForm1.FormClose(Sender: TObject; var CloseAction: TCloseAction);
begin
  //знищуємо змінну:
  sl.Free;
end;
```

Таким чином, ми гарантували виділення й звільнення пам'яті для змінної. Тепер зі змінної sl можна буде працювати в будь-якій процедурі або функції програми. Код обробки першої кнопки буде таким:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  //якщо тексту ні, відразу виходимо:
  if Edit1.Text = '' then exit;
  //додаємо рядок у кінець тексту:
  sl.Add(Edit1.Text);
  //очищаємо Edit1:
  Edit1.Text:= '';
end;
```

Як бачите, спочатку ми зробили перевірку - є чи текст у рядку Edit1? Якщо тексту ні, то ми виходимо із процедури, нічого не роблячи. Якщо текст існує, то спочатку ми додаємо його в змінну sl, а потім очищаємо Edit1 для нового тексту.

Код обробки другої кнопки ще простіший:

```
procedure TForm1.Button2Click(Sender: TObject);
```

```
begin
  ShowMessage(sl.Text); //виводимо весь текст на екран
end;
```

Тут нам не потрібно робити перевірку на існування тексту - навіть якщо тексту немає, буде виведене порожнє повідомлення. Якщо текст є, він буде виведений на екран. Код для третьої кнопки:

```
procedure TForm1.Button3Click(Sender: TObject);
begin
  //зберігаємо текст у файл:
  sl.SaveToFile('mytext.txt');
end;
```

Отут, я думаю, все зрозуміло - у такий же спосіб ми зберігали текст у файл за допомогою компонента TMemo. Якщо файлу немає, він буде створений, якщо є - перезаписаний. Зверніть увагу, ми вказали просто ім'я файлу. Тобто, файл буде створений у поточній папці - папці із програмою. Якщо ви хочете, щоб файл зберігався десь у конкретному місці, то разом з ім'ям потрібно вказувати й адресу, наприклад:

```
  sl.SaveToFile('C:/mytext.txt');
```

Код читання з файлу для останньої кнопки не набагато складніший, оскільки нам ще потрібно перевірити існування файлу:

```
procedure TForm1.Button4Click(Sender: TObject);
begin
  //читаємо текст із файлу, якщо він існує:
  if FileExists('mytext.txt') then
    sl.LoadFromFile('mytext.txt');
end;
```

Функція FileExists повертає True, якщо зазначений файл існує. У цьому випадку буде виконане читання з файлу. Якби такої перевірки не було, то спроба звертання до неіснуючого файлу привела б до помилки програми. Якщо при збереженні ви вказували ім'я файлу разом з його адресою, отут також потрібно вказати адресу. Спробуйте роботу програми, поекспериментируйте із кнопками, переконаєтеся, що файл створюється.

Однак, це ще не все. Фактично, змінна типу TStringList є масивом рядків, з яких складається текст. Як уже говорилося вище, звернутися

можна не тільки до всього тексту, але й до його конкретного рядка, указавши його індекс. Індексація рядків починається з нуля, властивість змінної Count містить загальну кількість рядків. Виправимо код для кнопки «**Вивести текст на екран**» - виведемо не весь текст, а тільки його останній рядок:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  //якщо взагалі текст є:
  if sl.Count > 0 then
    //то виведемо на екран останній рядок:
    ShowMessage(sl[sl.Count - 1]);
end;
```

Як бачите, тепер нам довелося зробити перевірку - а є чи текст? Якщо тексту ні, то спроба звертання до неіснуючого рядка викличе помилку програми. Оскільки індексація рядків починається з нуля, то індекс останнього рядка буде Count мінус одиниця. Отут ви можете помітити різницю звертання до тексту. Весь текст утримується в:

```
sl.Text
```

А перший рядок це:

```
sl[0]
```

Крім Count, тип TStringList має й інші, відомі по TMemo інструменти-методи:

Add - додати новий рядок у кінець тексту, наприклад:

```
sl.Add('Новий рядок');
```

Clear - очистити текст, наприклад:

```
sl.Clear;
```

Insert - вставити рядок у конкретне місце, указавши індекс рядка. Наприклад, вставити другий рядок:

```
sl.Insert(1, 'Новий другий рядок');
```

Delete - видалити зазначений рядок. У метод передається індекс видаляє рядки, що. Наприклад, видалити останній рядок:

```
sl.Delete(sl.Count - 1);
```

2. Робота з папками

В Lazarus є кілька корисних функцій і процедур для різної роботи з папками (каталогами), які ви можете застосовувати у своїх програмах. Вивчимо їх. При бажанні, можете створити простий додаток з єдиною кнопкою **TButton** на форми. Згенеруйте процедуру OnClick для неї, і в цій процедурі по черзі можете перевіряти розглянуті далі приклади.

GetCurrentDir - функція повертає адресу й ім'я поточної папки. Функція описана в модулі SysUtils, тобто, щоб використати функцію, потрібно переконатися, що SysUtils підключено в розділі uses. Приклад використання:

```
ShowMessage(GetCurrentDir);
```

GetDir - функція повертає поточну папку на зазначеному в параметрі диску. Є два параметри: номер диска (ціле число) і ім'я текстової змінної, куди буде записаний результат роботи функції. Майте на увазі, що нумерація дисків починається з 1, це буде диск A:. Перший жорсткий диск C: має номер 3. Якщо вказати номер диска 0, то буде виводиться інформація про поточну папку поточного диска. В Linux і інших Unix-подібних системах нумерація диска ігнорується. Функція описана в модулях System і SysUtils, для використання буде досить, якщо підключено один з них. Приклад використання:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  s: string;
begin
  GetDir(3, s);  //у змінну s одержали поточну папку на диску C:
  ShowMessage('Поточна папка на диску C: - ' + s);
end;
```

DirectoryExists - функція перевіряє, чи існує зазначена в параметрі папка. Якщо папка існує, функція повертає True, якщо немає - False. Якщо адреса папки не зазначена, то робиться перевірка на існування папки всередині поточної папки. Функція описана в модулі SysUtils, її варто використати, якщо програма буде звертатися до якоїсь папки: копіювати її або просто зробити поточною. Якщо папки немає - це викличе помилку програми, тому перевірка на існування в таких випадках обов'язкова. Приклад використання:

```
if not DirectoryExists('C:\Temp') then ShowMessage('Папки «C:\Temp» не існує')
```

else ShowMessage(‘Папка»C:\Temp» існує!!!’);

ChDir - процедура, робить поточною зазначену в параметрі папку. Процедура описана в модулях System і SysUtils, щоб її використати, досить підключити одину з них. Приклад використання:

ChDir(‘C:\Temp\111’); //змінили поточну папку
ShowMessage(‘Тепер поточна папка: ‘ + GetCurrentDir);

SetCurrentDir - функція, робить зазначену в параметрі папку поточної й повертає True у випадку успіху, або False, якщо відбулася помилка (наприклад, зазначеної папки не існує). Функція описана в модулі SysUtils. Приклад використання:

if SetCurrentDir(‘C:\Temp\1111’) then ShowMessage(‘Перемінили поточну папку’)
else ShowMessage(‘Помилка! Перемінити папку не вдалося’);

CreateDir - функція, створює зазначену в параметрі папку. У випадку успіху повертає True, у випадку невдачі - False. Невдача може трапитися, якщо відсутні права на створення папки, наприклад, диск доступний тільки для читання, або така папка вже існує. Функція описана в модулі SysUtils. Наступний приклад буде успішним при першому обігу (якщо звичайно, папку взагалі можна створити), і неуспішному при наступних, тому що така папка вже буде:

if CreateDir(‘C:\Temp\MyNewDir’) then
ShowMessage(‘Створили нову папку’)
else ShowMessage(‘Помилка! Створити папку не вдалося’);

ForceDirectories - функція, створює зазначену папку й всі відсутні папки. У випадку успіху повертає True, у випадку невдачі - False. Функція описана в модулі SysUtils. У наступному прикладі на диску C: буде створена папка **MyNewDir**, а в ній - папка **123**:

if ForceDirectories(‘C:\MyNewDir\123’) then ShowMessage(‘Папки створені’)
else ShowMessage(‘Помилка створення папок’);

RemoveDir - функція, видаляє зазначену в параметрі папку. Щоб видалення було успішним, папка повинна бути порожньою. Якщо в параметрі зазначений не абсолютний шлях (разом з адресою), функція намагається видалити зазначену папку всередині поточної папки. У

випадку успіху повертає True, у випадку невдачі - False. Функція описана в модулі SysUtils. Приклад використання:

```
if RemoveDir('C:\MyNewDir\123') then ShowMessage('Папка вилучена')
else ShowMessage('Помилка видалення папки');
```

RmDir - процедура, видаляє зазначену в параметрі папку. На відміну від RemoveDir, не повертає ніяких значень і не може видалити поточну папку. При спробі видалити неіснуючу папку викликає помилку програми, так що застосовувати потрібно тільки разом з DirectoryExists, щоб уникнути неприємних помилок. А краще завжди користуватися RemoveDir. Приклад використання:

```
RmDir('C:\MyNewDir\123');
```

Тема 23. Файли

1. Файли
2. Текстові файли
3. Типізовані файли
4. Нетипізовані файли

1.Файли

Файл - це область зовнішньої пам'яті ПК, що має ім'я.

Тобто, файл - це частина ПЗУ - Постійного Запам'ятовуючого Пристрою, яким може бути жорсткий диск, флэш-карта, CD або DVD. В операційних системах існують різні файлові структури, або, як ще говорять, таблиці розмищення файлів. Наприклад, в Windows це може бути FAT32 або NTFS. У цих таблицях імена файлів співвідносяться з реальною адресою на диску. На рівні машинних кодів, реальна адреса файлу являє собою ціле число - номер байта, з якого починається файл. Також у таблиці втримується загальна кількість байтів, зайнятих файлом. Але при програмуванні на мовах високого рівня, таких, як Об'єктний Паскаль, ми можемо користуватися символічними іменами файлів, які автоматично перетворяться операційною системою в реальні адреси й розміри цих файлів.

Файли бувають трьох типів:

1. **Текстові файли.** Це файли, призначені для роботи з текстом.

2. **Типізовані файли.** Такі файли можуть зберігати масив записів якогось одного, зазначеного типу. Це може бути дуже зручно для створення файлу з даними.

3. **Нетипізовані файли.** Це файли для побайтової роботи з даними будь-якого типу. Можуть використатися для копіювання або переносу файлів, тим більше що відсутність типу робить такі файли більш універсальними, сумісними з будь-яким існуючим типом. Крім того, у нетипізованих файлів обмін даними між пам'яттю й диском більше швидкісний, що важливо при обробці файлів великого розміру.

Щоб відкрити доступ до файлу, потрібно створити файлову змінну одним із трьох способів:

```
var
  f1: TextFile;  //текстовий файл
  f2: File of <тип>;  //типізований файл
  f3: File;  //нетипізований файл
```

Типізований файл може працювати з будь-якими типами Об'єктного Паскаля, крім файлів. Це може бути ціле або речовинне число, символ, рядок або навіть користувальницький тип даних, наприклад, запис (про записи - у наступній лекції). Наприклад, оголосити типізований файл, що працює із символами, можна так:

```
var
  My: File of Char;  //типізований символьний файл
```

Однак для роботи з файлами мало оголосити змінну файлового типу, потрібно ще цю змінну «прив'язати» до конкретного файлу. Робиться це за допомогою процедури AssignFile, у яку передається два параметри - ім'я файлової змінної, до якої «привязывается» файл, і ім'я існуючого файлу, з яким ми хочемо працювати. Якщо ім'я файлу зазначене без адреси, то процедура має на увазі, що файл перебуває в поточній папці. Якщо файлу ні, програма викличе помилку, тому рекомендується завжди перед зв'язуванням файлової змінної з файлом спочатку перевірити - чи існує файл? Робиться це за допомогою функції FileExists, з якої нам уже доводило працювати, і яка повертає True, якщо зазначений файл існує, і False у противному випадку. От приклад зв'язування файлової змінної з файлом:

```
var
  f1: TextFile;  //текстовий файл
```

```
begin
  if FileExists('c:\01\myfile.txt') then begin
    AssignFile(f1, 'c:\01\myfile.txt'); //зв'язали файлову змінну з файлом
    ... //подальша робота з файлом
  end; //if
```

Після того, як ми зв'язали файлову змінну з файлом, ми можемо звертатися до неї, як до файлу. Ми можемо відкрити її, уважати або записати інформацію, закрити. А після роботи з файлом його обов'язково потрібно закрити! Робиться це процедурою CloseFile, у яку передається всього один параметр - ім'я файлової змінної, наприклад:

```
CloseFile(f1);  //закрили файл, з яким була зв'язана файлова змінна f1
```

Під час роботи з файлом може виникнути **виняткова ситуація**. Увага! Такий термін ми зустрічаємо вперше, і він дуже важливий для розуміння. Що таке виняткова ситуація? Це будь-яка помилка програми, що може відбутися під час роботи. Наприклад, ви відкрили якийсь мережний файл і почали роботу з ним. А інший користувач у цей час взяв, так видалив цей файл. Або ви намагаєтеся працювати з файлом, що містить вірус. Як тільки ви звернулися до нього, ваш антивірус втручається, і видаляє файл, або переносить його в карантин. Або ж ваш файл «битий» - містить помилку, і не може прочитатися.

При спробі читання з неіснуючого файлу, або запису в нього, або при роботі з «битим» файлом відбудеться помилка й виникне виняткова ситуація. Якщо ви не обробите цю помилку, то ваша програма, швидше за все, намертво зависне. Вам доведеться викликати **Диспетчер завдань** Windows, і знімати завдання - примусово закривати вашу програму. Тому в будь-якій ситуації, коли є ризик виникнення виняткової ситуації, програміст ЗАВЖДИ повинен її обробити. Для цього існує блок try-finally-end:

```
try
  //блок коду, у якому може відбутися помилка
finally
  //код, що повинен виконатися в кожному разі, наприклад, код закриття файлу
end;
```

Миж службовими словами try-finally ви вписуєте потенційно небезпечний код. Тобто, код, що може викликати помилку. А миж finally-end указуєте той код, що повинен бути виконаний у кожному разі, навіть при виникненні помилки. От більше правильний приклад того, як потрібно зв'язувати файл зі змінної:

```
var
  f1: TextFile;  //текстовий файл
begin
  try
    AssignFile(f1, 'c:\01\myfile.txt'); //зв'язали файлову змінну з файлом
      …;  //подальша робота з файлом - читання/запис
  finally
    CloseFile(f1);  //по закінченні закриваємо файл
  end;  //try
```

Як бачите, тут спробу зв'язати файлову змінну з файлом, і подальшу роботу з ним ми уклали в блок try-finally. Оскільки блок обробляє виняткові ситуації, можна не робити додаткової перевірки на існування файлу. Чи вийшла відкриття файлу, і робота з ним, або відбулася помилка, у кожному разі буде виконане закриття файлу, що вбереже програму від краху.

Є й інший спосіб обробки виняткових ситуацій - блок try-except-end:

```
try
  //блок коду, у якому може відбутися помилка
except
  //код, що виконається в тому випадку, якщо в блоці миж try-except відбулася помилка
end;
```

Приклад роботи з файлом можна представити й так:

```
var
  f1: TextFile;  //текстовий файл
begin
  try
    AssignFile(f1, 'c:\01\myfile.txt'); //зв'язали файлову змінну з файлом
      …;  //подальша робота з файлом - читання/запис
    CloseFile(f1);  //по закінченні закриваємо файл
```

```
except
  ShowMessage('Увага! Відбулася помилка відкриття файлу.');
end;  //try
```

Яким образом обробляти виняткові ситуації - вирішувати вам. Залежно від ситуації, іноді буває удобней перший спосіб, іноді - другої.

Після зв'язування файлової змінної з файлом, цей файл потрібно **ініціювати**, тобто, указати напрямок передачі даних - з файлу (читання), у файл (запис), і в обох напрямках. Якщо ми відкриваємо файл для читання, нам варто скористатися стандартною процедурою Reset:

```
Reset(f1);
```

При цьому покажчик установлюється в початок файлу, у нульову позицію.

Якщо нам потрібно створити новий файл, або перезаписати існуючий, використається процедура Rewrite:

```
Rewrite(f2);
```

Якщо файл уже існував, він буде перезаписаний без яких або попереджень. Якщо ж нам потрібно відкрити для запису існуючий файл, не видаляючи інформації, що втримується в ньому, скористаємося процедурою Append:

```
Append(f1);
```

Файл буде відкритий для запису, а покажчик переміститься в кінець файлу. Однак, процедура Append застосовна тільки до текстових файлів, тобто до змінних типу TextFile.

Під час роботи з файлами можна використати функції Bof і Eof - перша повертає істину, якщо покажчик перебуває на початку файлу, друга - якщо наприкінці. Обидві функції мають параметр - файлову змінну:

```
if Eof(f1) then ShowMessage('Досягнуть кінець файлу.');
```

Є ще один спосіб перевірки існування файлу, і взагалі обробки помилок виводу-вводу-висновку у файл. Це - функція IOResult, що повертає нуль при відсутності помилок виводу-вводу-висновку, і номер помилки в противному випадку. Але використати цю функцію просто так не вийде. Lazarus автоматично обробляє помилки виводу-

вводу-висновку. Перш ніж використати IOResult, потрібно дати процесору команду відключити автоматичну обробку цих помилок. Для цього існує директива процесора {$I-}. Потім ми виконуємо небезпечну ділянку коду, де може відбутися помилка, після чого знову включаємо автоконтроль операцій виводу-вводу-висновку директивою {$I+}. Потім викликаємо IOResult. Якщо операція відбулася успішно, то функція поверне нуль. Приклад:

```
var
  f1: TextFile;
begin
  AssignFile(f1, 'MyText.txt');  //зв'язали файл зі змінної
  {$I-}  //відключили автоконтроль виводу-вводу-висновку
  Reset(f1);  //намагаємося відкрити файл для читання
  {$I+}  //знову включили автоконтроль виводу-вводу-висновку
  if IOResult <> 0 then
    ShowMessage('Увага! Відбулася помилка відкриття файлу.');
```

2. Текстові файли

Ці файли зв'язуються зі змінної, що має тип TextFile, і призначені для зберігання й обробки тексту. Фактично, у Паскале текст - це набір рядків різної довжини.

На початку файлу встановлюється спеціальна системна ознака BOF (Begin Of File) - початок файлу. Наприкінці кожного рядка ставиться спеціальна ознака кінця рядка EOLN (End Of Line) - кінець лінії, рядка. Це два системні, невидимих символи - переклад каретки й новий рядок, ми їх можемо виразити, як символи #13 і #10. Наприкінці файлу встановлюється ознака EOF (End Of File) - кінець файлу.

Для доступу до записів застосовуються процедури Read і Readln для читання, Write і Writeln для запису. Ми вже знайоми із цими процедурами по лекции №21, де ми вивчали консольні додатки, однак з текстовими файлами ці процедури працюють трохи інакше. У всіх цих процедурах першим параметром іде файлова змінна, а другим - список аргументів. Аргументами можуть бути одна або трохи змінних строкового, символьного або числового типу. Для приклада, створимо простий додаток, що дозволяє працювати з текстовим файлом, заносячи в нього построчно текст із рядка TEdit.

Відкрийте **Lazarus** з новим проєктом. В Caption форми напишіть «Робота з текстовим файлом». Проєкт у нас пробний, так що імена форми, проєкту й компонентів ми зминювати не будемо. Просто

збережете проєкт у папку **23-01**. Встановіть на форму компонентів TEdit і подовжите його, щоб користувач миг увести довгий рядок. Очистите в компонента властивість Text. Нижче встановите дві кнопки, в Caption першої напишіть «Додати у файл», другий - «Уважати з файлу». Розтягніть їх, щоб текст на кнопках не здавався прилиплої до країв. Ще нижче встановите TMemo. Відкрийте редактор властивості Lines і видалите відтіля текст. Розтягніть компонент по розмирі Edit1.

У результаті, у вас повинна вийти приблизно така форма:



Рис. 23.1. Форма проєкту

З підготовчою частиною закінчили, займемося програмуванням кнопок. Згенеруйте подію OnClick для першої кнопки. Її код буде таким:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  tf: TextFile;  //текстовий файл
  s: String;
begin
  //якщо користувач не ввів рядок в Edit1, просто вийдемо:
  if Edit1.Text = ‘‘ then exit;
  //інакше в s одержуємо текст:
  s:= Edit1.Text;
  //потім зв’язуємо файлову змінну tf з файлом mytext.txt:
  AssignFile(tf, ‘mytext.txt’);
  //далі може виникнути виняткова ситуація, тому
  //укладемо небезпечний код у блок try-finally-end:
```

```
    try
      //якщо файлу немає (кнопку нажали перший раз), створимо його:
      if not FileExists('mytext.txt') then Rewrite(tf)
      //інакше відкриємо для запису, установивши покажчик у кінець файлу:
      else Append(tf);
      //отут просто записуємо рядок у файл:
      Writeln(tf, s);
      //очищаємо Edit1, щоб користувач бачив, що подія відбулася:
      Edit1.Text:= '';
    finally
      CloseFile(tf); //закриваємо файл
    end;
  end;
```

Коментарі досить докладні, щоб зрозумити код. Помичу тільки пару речей. Якщо користувач нічого не ввів в Edit1, то нам немає потреби відкривати файл і зберігати в нього порожній рядок, тому ми просто виходимо із процедури, нічого не роблячи:

```
    if Edit1.Text = '' then exit;
```

Коли ми відкриваємо файл, ми спочатку робимо перевірку на його існування. Файлу може не бути у двох випадках: користувач нажав на кнопку перший раз, або, попрацювавши з файлом, і не закриваючи програми, він видалив цей файл. У цьому випадку процедурою Rewrite ми створюємо новий файл. Якщо ж файл уже існує (кнопка натискалася, або програма відкривається не в перший раз, і файл залишився з минулих сеансів), то процедурою Append ми відкриваємо його для запису, перемищаючи покажчик у кінець файлу. Далі все просто - ми записуємо рядок тексту у файл, очищаємо Edit1 від старого тексту й закриваємо файл. Код другої кнопки буде схожим:

```
  procedure TForm1.Button2Click(Sender: TObject);
  var
    tf: TextFile;  //текстовий файл
    s: String;
  begin
    //якщо файлу ні, просто виходимо:
    if not FileExists('mytext.txt') then exit;
    //інакше спочатку очищаємо Memo1:
```

```
  Memo1.Clear;
  //зв'язуємо файлову змінну tf з файлом mytext.txt:
  AssignFile(tf, 'mytext.txt');
  //далі може виникнути виняткова ситуація, тому
  //укладемо небезпечний код у блок try-finally-end:
  try
    Reset(tf); //відкрили файл для читання, покажчик на початку файлу
    //робимо, поки не кінець файлу:
    while not Eof(tf) do begin
      Readln(tf, s); //читаємо в s черговий рядок
      Memo1.Lines.Add(s);  //додаємо цей рядок в Memo1
    end;
  finally
    CloseFile(tf); //закриваємо файл
  end;
end;
```

По коментарях зрозумило, яким образом відбувається читання. Зверніть увагу, для запису ми скористалися процедурою Writeln, а не Write, а при читанні - Readln, а не Read. Це позбавило нас від необхідності піклуватися про вставку й пошук символів кінця рядка, рекомендую вам завжди використати такий, більше зручний спосіб. Весь алгоритм зчитування рядків ми помистили в блок while-do і зчитували текст рядок за рядком, поки функція Eof не повернула True, тобто, поки файл не закінчився.

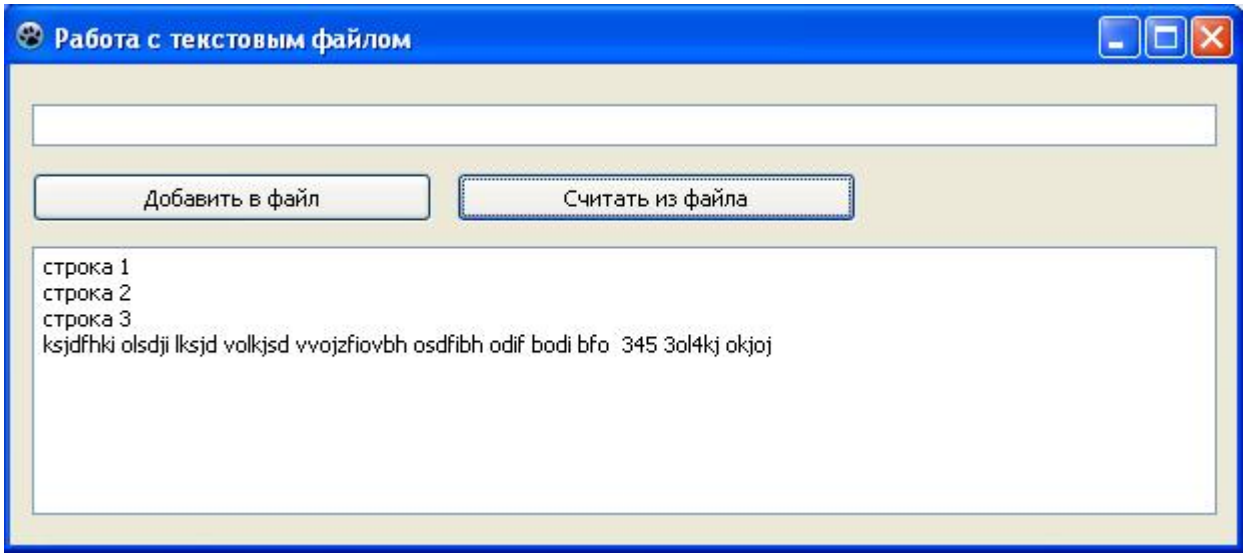


Рис. 23.2. Програма в дії

3. Типізовані файли

Такі файли призначені для обробки інформації якогось певного типу. Це може бути й символ, і число. Якщо це рядок, то її потрібно повідомляти з певною довжиною, наприклад, так:

```
var
  s: string[50];
```

Крім того, файл може прийняти користувальницький тип даних, наприклад, запис (запису див. у наступній лекції). Оскільки компілятор знає довжину типу в байтах, то він може організувати доступ до будь-якого елемента типізованого файлу по його порядковому номері. Як тільки файл відкритий, покажчик установлюється в початок файлу й указує на перший елемент із номером 0. Після читання або запису покажчик зрушується до наступного елемента. Це було б неможливо, якби компілятор заздалегідь не знав розмир кожного зчитувального або записуваного елемента, тому й рядок повинна мати строго певний розмир.

До типізованих файлів застосовні наступні функції й процедури:

FilePos - функція приймає параметр - файлову змінну, і повертає позицію покажчика, тобто, номер елемента, що буде оброблятися при наступному читанні/запису. Приклад:

```
FilePos(f1);
```

FileSize - функція, що також приймає як параметр файлову змінну, і повертає кількість елементів у файлі. Приклад:

```
FileSize(f2);
```

Read - процедура читання даних з типізованого файлу. Як параметри вказується файлова змінна, і одна або трохи змінних з таким же типом даних, як у файлу. У ці змінні буде лічений один або трохи (по кількості змінних) елементів, при цьому покажчик буде зрушуватися до наступного елемента. Приклад:

```
Read(f1, MyRealPerem);
```

Write - процедура запису в типізований файл. Як і в Read, в Write є параметри - файлова змінна, і одна або трохи змінних з таким же типом даних, як у файлу. Із цієї змінної (цих змінних) будуть братися дані, які будуть записані у файл. Після кожного запису покажчик перемищається. Приклад:

Write(f1, MyRealPerem);

Seek - процедура зсуву покажчика. Процедурі потрібно вказати файлову змінну й номер елемента, до якого потрібно перемістити покажчик. Приклади:

```
Seek(f, 0);  //покажчик у початок файлу
Seek(f, 1);  //покажчик на другий елемент
Seek(f, FileSize(f));   //покажчик у кінець файлу
```

Ще раз помичу, що такі процедури, як Readln, Writeln і Append працюють **тільки з текстовими файлами**, у типізованих або нетипізованих файлах їх використати не можна.

Вивчимо роботу з типізованими файлами на простому прикладі.

Відкрийте **Lazarus** з новим проєктом. В Caption форми напишіть «Типізований файл» і збережете проєкт у папку **23-02**, не миняючи імен форми, модуля й проєкту.

На форму встановите компонент TListBox. Отриманий ListBox1 небагато витягніть по висоті, щоб у ньому вмистилися 10 чисел. Праворуч Встановіть одну під інший, дві прості кнопки TButton, в Caption яких напишіть, відповідно, «Сформувати файл» і «Читати з файлу». У вас повинна вийти приблизно така форма:

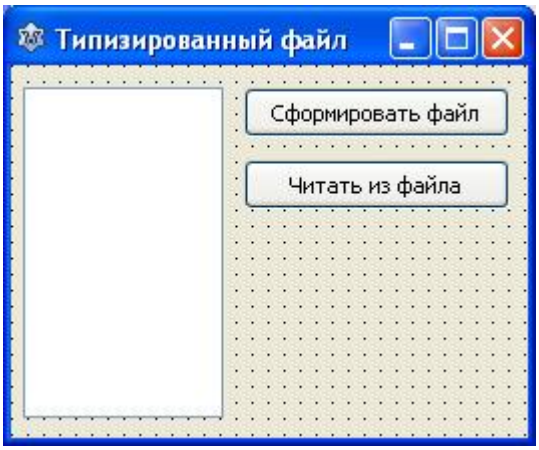


Рис. 23.3. Форма проєкту

Ідея така. При натисканні на кнопку «**Сформувати файл**» ми за допомогою функції Random будемо формувати 10 випадкових чисел, від 0 до 1000, і записувати їх у типізований файл. Файл у нас буде мати тип Integer, тобто, під кожний елемент у файлі буде виділено 4 байти. Таким чином, буде сформовано 10 елементів по 4 байти, у результаті вийде файл розміром 40 байт.

При натисканні на кнопку «**Читати з файлу**» ми будемо по одному зчитувати ці елементи, і додавати їх в ListBox1. Таким чином, ми переконаємося, що запис у типізований файл, і читання з нього відбуваються без помилок. Код першої кнопки наступний:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  f: File of integer;  //файл типу integer
  i: integer;  //лічильник для циклу
begin
  AssignFile(f, 'mytypefile.dat');
  Rewrite(f);
  for i:= 1 to 10 do  //робимо 10 разів
    Write(f, Random(1000));  //записуємо у файл випадкове ціле число
  CloseFile(f);
end;
```

Код багато в чому вам знайомий, щоб ви змогли з ним розібратися без докучливих коментарів. Код другої кнопки:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  f: File of integer;
  k: integer;  //для одержання значень елементів
begin
  //виходимо із процедури, якщо файлу немає:
  if not FileExists('mytypefile.dat') then exit;
  ListBox1.Clear;
  AssignFile(f, 'mytypefile.dat');
  //Відкриваємо для читання:
  Reset(f);
  //робимо від першого до останнього елемента:
  while not eof(f) do begin
    Read(f, k); //зчитуємо черговий елемент у перем. k
    ListBox1.Items.Add(IntToStr(k));      //записуємо це значення в ListBox
  end;
  CloseFile(f);
end;
```

Код, я думаю, також не має потреби в додаткових коментарях. Однак все-таки дещо помичу. Оскільки файл типізований, інформація в нього буде записана не в текстовому, а у двійковому виді, тому, відкривши отриманий файл простим текстовим редактором, ви не побачите там тих же чисел, що були у вас в ListBox1. Властиво, це навіть добре - простий текстовий файл із даними користувач може, заради інтересу, і зіпсувати нерозумною спробою редагування, а от у двійковий файл він лізти побоїться.

4. Нетипізовані файли

Такі файли асоціюються зі зминними типу File, і можуть обробляти побайтово дані будь-якого типу, тому їх зручно використати для копіювання/переносу файлів. Нетипізовані файли не підтримують функції читання-запису Read, Readln, Write і Writeln. Замість них типізованим файлам доступні більше швидкісні функції BlockRead і BlockWrite. Ці функції дозволяють робити читання-запис блоками, використовуючи для цього зминн-буфер.

Відкривати файл для читання або запису потрібно функціями Reset і Rewrite, але отут теж є відминність: крім файлової змінної цим функціям можна вказувати довжину запису в байтах. Якщо довжина запису не зазначена, приймається довжина за замовчуванням 128 байт. Однак звичайно вказують довжину 1 байт, що дозволяє обробляти будь-які, навіть сами маленькі файли, наприклад:

```
Reset(f1, 1);
Rewrite(f2, 1);
```

Крім цих розходжень, нетипізованим файлам доступні всі інші процедури й функції, що й типізованим. Вивчимо роботу з нетипізованими файлами на прикладі, для цього створимо програму, що виконує копіювання файлу.

Відкрийте **Lazarus** з новим проєктом. Ім’я форми змінимо на fMain (це буде корисна програма, так що імена будемо давати осмислені), в Caption напишемо «Копіювання файлу», BorderStyle установимо в bsDialog, а Position - в poDesktopCenter. Збережемо проєкт у папку **23-03** під ім’ям **FileCopier**, а модулю головної форми дамо ім’я Main.

Встановіть на форму митку TLabel, в Caption якої напишіть «Виберіть файл, якому потрібно копіювати:». Тепер перейдіть на вкладку **Misc Палітри компонентів**, знайдіть на ній і встановите на форму компонентів TFileNameEdit - це гібрид TEdit і діалогу відкриття файлу, він полегшить нам взаємодія з користувачем. Нам оведеться

звертатися до компонента в коді, скоротите його ім’я (властивість Name) до FNE.

Нижче встановите ще одну митку, в Caption якої напишіть «Укажіть папку, куди копіювати файл:». Ще нижче встановите компонент TDirectoryEdit з тієї ж вкладки **Misc** - цей компонент полегшить нам вибір папки, куди будемо робити копію. Скоротите його ім’я до DE.

І наостанок, встановіть в нижній частині форми просту кнопку **TButton**, в Caption якої напишіть «Виконати копіювання». Кнопка в нас одна, так що її можна не перейменовувати.

Змініть розміри форми й компонентів, щоб у вас вийшло приблизно наступне:

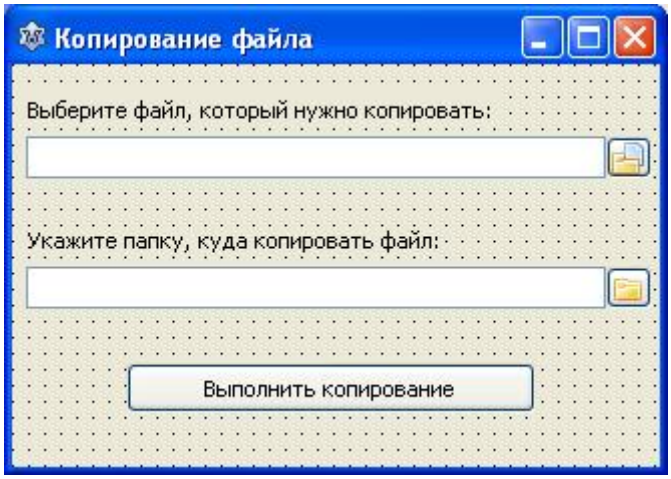


Рис. 23.4. Форма проєкту

З підготовчою роботою закінчили, перейдемо до кодування. Весь код нам потрібно буде робити в події OnClick кнопки, згенеруйте його. От цей код:

```
procedure TfMain.Button1Click(Sender: TObject);
var
  fIn, fOut : File; //нетипізовані файли: исходник і копія
  NumRead, NumWritten : Word; //кількість лічених і записаних байт
  Buf : Array[1..2048] of byte;  //буфер
begin
  //якщо файл не обраний, робимо повідомлення й виходимо:
  if FNE.Text = ‘‘ then begin
    ShowMessage(‘Увага! Потрібно вказати або вибрати копійований файл.’);
    FNE.SetFocus;
    Exit;
```

```
    end;
    //якщо не обрана папка, робимо повідомлення й виходимо:
    if DE.Text = ‘‘ then begin
      ShowMessage(‘Увага! Потрібно вказати або вибрати папку для копії.’);
      DE.SetFocus;
      Exit;
    end;
    //починаємо роботу з файлами:
    try
      AssignFile(fIn, UTF8ToSys(FNE.FileName));
      AssignFile(fOut,UTF8ToSys(DE.Directory + ‘\’ + ExtractFileName(FNE.FileName)));
      Reset (fIn,1);
      Rewrite (fOut,1);
      //змінимо курсор на піскові годинники:
      Screen.Cursor:= crHourGlass;
      Repeat
        BlockRead (fIn,buf,Sizeof(buf),NumRead);
        BlockWrite (fOut,Buf,NumRead,NumWritten);
      Until (NumRead=0) or (NumWritten<>NumRead);
    finally
      closeFile(fIn);
      closeFile(fOut);
      //змінимо курсор на звичайний вид:
      Screen.Cursor:= crDefault;
      ShowMessage(‘Копіювання завершене!’);
    end;
  end;
```

У коді є дещо нове, давайте розберемо його детальніше. Спочатку ми оголосили змінні fIn і fOut, які мають тип File. Ці змінні ми асоціюємо з файлами: fIn - з вихідним файлом, а fOut - зі створюваною копією.

Далі, ми створили змінн-лічильники NumRead і NumWritten типу Word. У ці змінні буде заноситися кількість реально порахованих і записаних байт.

Також ми створили буфер - масив, у який можна записати до 2048 байт. У принципі, масив можна зробити будь-якого розміру, хоч 1 байт, але цифру 2048 взято з довідкової системи самого Lazarus.

Далі йде перевірка - чи вказав користувач потрібні нам дані? Якщо він залишив порожній рядок з файлом - оригіналом, або з папкою, куди потрібно робити копію, ми виводимо відповідне повідомлення, і методом SetFocus виділяємо потрібний компонент (той, котрий пропустив користувач), після чого виходимо із процедури.

Якщо користувач вказав всі дані, то далі починається робота з файлами. Ми асоціюємо файли з файловими змінними:

```
AssignFile(fIn, UTF8ToSys(FNE.FileName));
AssignFile(fOut,UTF8ToSys(DE.Directory + '\' +
ExtractFileName(FNE.FileName)));
```

Зверніть увагу: у властивості FileName компонента TFileNameEdit перебуває адреса й ім'я обраного користувачем файлу. У властивості Directory компонента TDirectoryEdit перебуває адреса й ім'я обраної користувачем папки. Але тут ми додатково обробили ці рядки функцією UTF8ToSys. У принципі, цього можна було й не робити, але тоді програма викликала б помилку, якби в адресі або імені файлу (папки) були неанглійські символи. Функція UTF8ToSys перетворить символи UTF8 у системне кодуваннння, що потрібно процедурі AssignFile. Тепер ім'я файлу й адреса може бути хоч кирилицею, хоч китайськими ієрогліфами - програма не викличе помилки. Рекомендую щоб уникнути можливих помилок завжди обробляти асоційовані файли цією функцією. Функція UTF8ToSys описана в модулі FileUtil, що за замовчуванням уже підключений до проєкту в розділі uses.

Далі, ми відкрили оригінал для читання, і створили копію, вказавши, що за раз будемо обробляти по одному байті:

```
Reset (fIn,1);
Rewrite (fOut,1);
```

А далі йде цікавий код, що нам ще не зустрічався:

```
Screen.Cursor:= crHourGlass;
```

У будь-якому графічному додатку є системний об'єкт Screen - екран. У цього об'єкта є властивість Cursor, що відповідає за зовнішній вигляд покажчика миші. Якщо привласнити цій властивості значення crHourGlass, то курсор буде мати вигляд пісочного годинника. Так

роблять, коли програма виконує якусь тривалу дію, і потрібно показати користувачеві, що комп'ютер не завис. Звичайно, якщо файл невеликий, він скопіюється миттєво, ви навіть оком не встигнете моргнути. Але якщо файл великий, наприклад фільм розміром у півтора^-два гігабайта, то копіювання такого файлу займе якийсь час. Тому ми й змінили зовнішній вигляд курсору. У самому кінці рядком

```
Screen.Cursor:= crDefault;
```

ми повертаємо курсору звичайний вид. Якщо вам цікаво, які ще види курсору бувають, виділите форму й подивитеся властивість Cursor - там досить значний список можливих значень.

Далі ми виконуємо читання-запис:

```
Repeat
  BlockRead (fIn,buf,Sizeof(buf),NumRead);
  BlockWrite (fOut,Buf,NumRead,NumWritten);
Until (NumRead=0) or (NumWritten<>NumRead);
```

Цикл Repeat...Until буде виконуватися, поки кількість лічених байт не стане рівним нулю, що означає, що файл закінчений, або поки кількість записаних і лічених байт не стане відрізнятися (відбулася яка те помилка читання-запису). Усередині циклу ми спочатку читаємо дані:

```
BlockRead (fIn,buf,Sizeof(buf),NumRead);
```

Як бачите, процедура читання BlockRead має чотири параметри:

1) Файлова змінна.

2) Буфер, куди будуть лічені дані.

3) Кількість чита байт. У нас зазначено Sizeof(buf). Функція SizeOf повертає розмир змінної, тобто в цьому випадку це буде 2048 байт.

4) Змінна, куди буде записана кількість реальна прочитаних байт, адже їх може бути менше, ніж 2048.

Тепер про запис:

```
BlockWrite (fOut,Buf,NumRead,NumWritten);
```

Процедура копіювання BlockWrite також має чотири параметри:

5) Файлова змінна.

6) Буфер, звідки будуть братися дані для запису.

7) Змінна з кількістю даних, які потрібно записати. Процедура BlockRead записала в змінну NumRead кількість реально лічених байт, це кількість ми й будемо писати.

8) Змінна з кількістю реально записаних байт.

Щораз, при читанні або записі, покажчик установлюється в нову позицію, так що нам спеціально його рухати не потрібно. Припустимо, що ми вже всі байти вважали. Тоді наступного разу при спробі вважати нову порцію в змінну NumRead потрапить 0 байт, оскільки файл скінчився, читати більше нема чого.

Після копіювання ми закриваємо файли, робимо покажчик миші звичайного виду й виводимо повідомлення, що файл скопійований. Збережете проєкт, запустіть його й спробуйте скопіювати файл. Якщо ви всі зробили правильно - файл скопируется без проблем. Перенос файлу - це те ж саме копіювання, тільки з наступним видаленням оригіналу.

Тема 24. Записи й варіант. Сітка рядків TStringGrid

1. Запис - користувальницький тип даних
2. Варіант
3. Сітка рядків TStringGrid

1. Запис - користувальницький тип даних

У Паскале передбачені не тільки стандартні типи, можна використати й нові типи даних, певні користувачем. Один з таких типів - запис.

Запис (анг. record) - це структура даних, певна користувачем з різних стандартно-типових полів.

По суті, запис є мініоб'єктом, що має властивості (поля), але не має методів і подій. Ще запис можна зрівняти з масивом - у ній також зберігаються різні дані, об'єднані в одній змінної, тільки на відмину від масиву, ці дані можуть бути різнотипними.

Запису часто застосовують для збереження й завантаження різних даних, наприклад, настроювань програми. За допомогою записів можна навіть зробити невелику базу даних, використовуючи типовий файл, що ми й проробимо в наступній лабораторній роботі.

Використання записів відбувається у два етапи: спочатку оголошується сам запис, потім оголошується змінна, котрої

привласнюють тип цього запису. До речі, тип запису можна привласнити не тільки змінної, але й масиву, і типізованому файлу.

Записи оголошуються в розділі type перед розділом var, і можуть мати як локальну область видимості (запис оголошений усередині підпрограми), так і глобальну. Синтаксис оголошення наступний:

```
type
  <ім'я_запису> = record
    <поле1>: <тип_поля>;
    <поле2>: <тип_поля>;
    …;
    <поле>: <тип_поля>;
  end;
```

Тут, ім'я_запису - ім'я, яким користувач називає новий створений тип даних. Поле - ім'я поля запису, фактично, це змінна усередині інший змінної. Тип_поля - один зі стандартних типів. Як уже говорилося, повідомляти тип string у записі не забороняється, однак використати такий запис у типізованому файлі не вийде. Обумовлено це тим, що при створенні типізованого файлу, компілятор повинен точно знати, скільки байт на кожний запис він повинен відвести у файлі. Рядок string є динамичним символьним масивом, тобто, заздалегідь довідатися її довжину неможливо. Однак, можна вирішити й цю проблему, якщо поставити обмежник на рядок, наприклад, string[50] - рядок з 50 символів. У цьому випадку, типізований файл можна створити. І якщо рядок буде порожня, або в ній до 50 не буде вистачати символів, то частина цього запису у файлі просто виявиться порожній, хоча місце під неї буде виділено. Правда, використання кирилиці в кодуванні UTF-8 створює деякі труднощі, але й вони розв'язувані.

Після того, як запис оголошений, потрібно створити змінну, привласнивши їй тип цього запису. Робиться це звичайним образом, у локальному або глобальному розділі var:

```
var
  NewPerem: <ім'я_запису>;
```

Після чого, можна звертатися до окремого поля цього запису, розділяючи його від імені зминною крапкою, наприклад:

```
NewPerem.Pole1:= 10;
```

Давайте освоїмо роботу із записами на прикладі програми, що зберігає свої настроювання, такі як положення й розміри форми, стан деяких компонентів, у типізованому файлі, а при наступному завантаженні програма зчитує ці настроювання й підбудовується під них.

Відкрийте **Lazarus** з новим проєктом. Відразу домовимося, що оскільки проєкт дуже прості й пробний, то імена форми, модуля, проєкту й компонентів ми залишимо за замовчуванням. Відразу ж збережете проєкт у папку **24-01**. Тепер, на форми нам будуть потрібні:

- Митка TLabel;
- Рядок TEdit;
- Кнопка TButton;
- Прапорець TCheckBox.

В Caption мітки напишіть «Заголовок форми:». Очистите Text в Edit1. В Caption кнопки напишіть «Застосувати». А із прапорцем можна нічого не робити. У результаті у вас повинна вийти приблизно така форма:

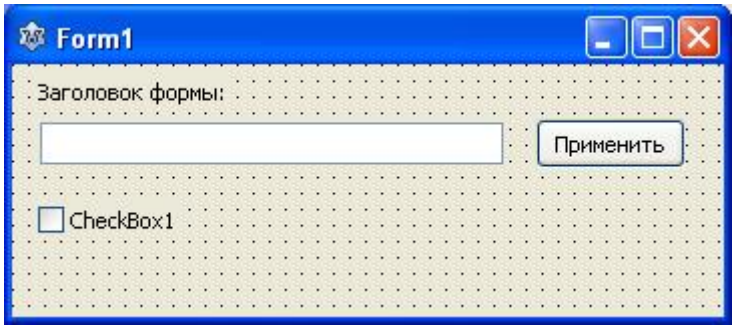


Рис. 24.1. Форма проєкту

Тепер саме головне - нам потрібно описати запис, у який потім збережемо параметри форми. Для цього потрібно створити розділ type ПЕРЕД глобальним розділом var, тим самим, у якому описана змінна форми:

```
var
  Form1: TForm1;
```

І не бентежтеся, що вище розділ type уже був, таких розділів може бути багато. Код нашої вставки наступний:

```
type
  myForm = record  //запис
    Left: integer;  //ліва границя форми
    Top: integer;  //верхня
    Height: integer;  //висота форми
```

```
    Width: integer;  //ширина
    Caption: string[100];  //заголовок
    Checked: boolean;  //стан прапорця CheckBox1
    wsMax: boolean;  //вікно розгорнуте?
  end; //record
```

Як бачите, всередині пустого type ми описали новий, користувальницький (наш) тип даних - запис. Тепер нам потрібна змінна цього типу, з якою ми будемо працювати. Нашій програмі потрібно зберігати свої налаштування при виході, робити це краще в події OnClose форми. Але їй також потрібно й зчитувати ці налаштування при завантаженні, робити це будемо в події OnCreate форми. Оскільки змінна нам буде потрібна й там, і там, то виходить, опишемо її в глобальному розділі var, відразу ПІСЛЯ опису змінної Form1:

```
var
  Form1: TForm1;
  My: myForm;  //наш змінн-запис
```

Тепер займемося кодуванням. Насамперед, нам потрібно «навчити» програму змінювати свій заголовок. Для цього згенеруйте подію OnClick для кнопки «**Застосувати**», його код:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if Edit1.Text <> ‘‘ then Form1.Caption:= Edit1.Text;
end;
```

Отут все просто: ми перевіряємо - чи не порожній рядок Edit1, і якщо не порожній, то привласнюємо йому текст властивості Caption форми, тим самим, змінюючи заголовок вікна. Підемо далі. А далі нам потрібно навчити програму зберігати свої налаштування. Для цього виділіть форму, клацнувши по ній у будь-якому вільному місці. Тепер перейдіть на вкладку **Події Інспектори об’єктів**, і подвійним клацанням згенеруйте подію форми OnClose, його код:

```
procedure TForm1.FormClose(Sender: TObject; var CloseAction:
TCloseAction);
var
  f: file of myForm;  //типізований файл для збереження даних
begin
```

```
  //спочатку налаштуємо змінн-запис:
  //не зберігаємо розміри й положення, якщо вікно wsMaximized:
  if not (Form1.WindowState = wsMaximized) then begin
    My.Left:= Form1.Left;
    My.Top:= Form1.Top;
    My.Height:= Form1.Height;
    My.Width:= Form1.Width;
    My.wsMax:= False;
  end
  else My.wsMax:= True;
  //інші налаштування:
  My.Caption:= Form1.Caption;
  My.Checked:= CheckBox1.Checked;
  //тепер створюємо або перезаписуємо файл налаштувань:
  try
    AssignFile(f, 'MyProg.dat');
    Rewrite(f);
    Write(f, My); //записуємо у файл дані із запису
  finally
    CloseFile(f);
  end;
end;
```

Давайте розберемо цей код. У розділі var ми описали змінну f, що буде асоціюватися з типізованим файлом, що має тип нашого запису. Бачите, ми привласнили наш тип і змінній, і типізованому файлу!

Далі нам потрібно зберегти в змінн-запис налаштування нашої форми, але отут є одне АЛЕ: якщо вікно програми було розгорнуто, то такі дані, як ліва й верхня границя вікна, його висота й ширина стають недійсними. Тому ми пішли на хитрість: якщо вікно не було розгорнуто, то ми зберігаємо ці параметри й, крім того, полю wsMax привласнюємо False, тим самим вказуючи, що вікно не було розгорнуто. Якщо ж вікно було розгорнуто, то ми привласнюємо цьому полю True, не торкаючи границі вікна:

```
  if not (Form1.WindowState = wsMaximized) then begin
    My.Left:= Form1.Left;
    My.Top:= Form1.Top;
    My.Height:= Form1.Height;
    My.Width:= Form1.Width;
```

```
  My.wsMax:= False;
 end
 else My.wsMax:= True;
```

Як видно з коду, ми звертаємося до окремих полів запису через крапку:

```
  My.Left
```

Інші параметри (заголовок вікна й стан прапорця) нам потрібно зберігати кожного разу:

```
 My.Caption:= Form1.Caption;
 My.Checked:= CheckBox1.Checked;
```

Далі ми працюємо з файлом, створюючи або перезаписуючи його. У файл ми записуємо одн-єдиний запис - нашу змінну My, у якій уже зберігаються всі потрібні налаштування, після чого файл закриваємо. Забігаючи вперед, скажу, що якщо ви спробуєте відкрити такий файл редактором текстів, то побачите в ньому двійковий код:

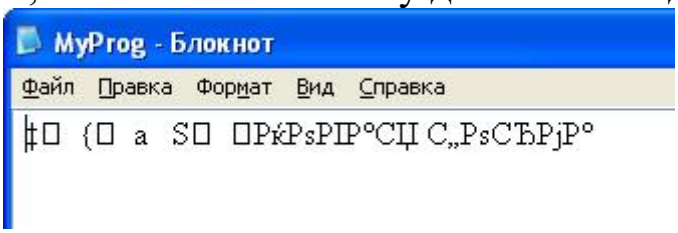


Рис. 24.2. Вміст файлу з налаштуваннями

Недосвідчений користувач не ризикне намагатися правити такий файл, а якщо ви випадково чогось у ньому зміните, і програма перестане завантажуватися, то просто видалите файл - без нього програма завантажиться з налаштуваннями за замовчуванням, а при наступному закритті файл **MyProg.dat** буде створений знову.

Тепер потрібно навчити програму зчитувати налаштування з файлу, і застосовувати їх. Згенеруйте подію форми OnCreate, його код:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  f: file of myForm;  //типізований файл для збереження даних
begin
  //якщо файлу немає, просто виходимо:
  if not FileExists('MyProg.dat') then exit;
```

//інакше відкриваємо файл і зчитуємо з нього налаштування в запис:

```
    try
      AssignFile(f, 'MyProg.dat');
      Reset(f);
      Read(f, My); //вважати дані в змінну типу запис
    finally
      CloseFile(f);
    end;
    //тепер налаштування потрібно застосувати до форми:
    //якщо вікно було розгорнуто, розвертаємо його, інакше набудовуємо:
    if My.wsMax then Form1.WindowState:= wsMaximized
    else begin
      Form1.Left:= My.Left;
      Form1.Top:= My.Top;
      Form1.Height:= My.Height;
      Form1.Width:= My.Width;
    end;
    //інші налаштування:
    Form1.Caption:= My.Caption;
    CheckBox1.Checked:= My.Checked;
  end;
```

Спочатку ми перевіряємо - є чи файл. Якщо немає (програма відкривається вперше), то ми виходимо, нічого не зчитуючи й не змінюючи налаштувань. У цьому випадку, всі налаштування будуть, як при розробці форми.

Коментувати роботу з файлом я не буду - все має бути гранично ясно по минулій темі й по попередньому коді збереження файлу. У наш змінн-запис My вважалися налаштування, збережені у файлі, і нам залишилося тільки застосувати їх до форми. Робиться це як при збереженні, але у зворотному напрямку - не в запис із форми, а у форму із запису. Також, якщо вікно було розгорнуто, розвертаємо його, інакше застосовуємо налаштування положення й розмірів форми.

Тепер збережіть проєкт, запустіть його на виконання й спробуйте змінити положення й розміри вікна, розвертати його - всі ці налаштування при виході повинні зберігатися, а при наступному запуску - відновлюватися.

2.Варіант

В Lazarus існує ще один цікавий тип даних - variant. Варіантний тип був розроблений в Object Pascal для підтримки роботи з механізмом OLE (англ. Object Link and Embedding - впровадження й зв'язок об'єктів). Навіть якщо ви вперше чуєте про такий механізм, ви з ним напевно знайомі, і не раз ним користувалися. Щораз, коли ви копіюєте таблицю MS Excel в MS Word, або картинку з Paint у той же MS Word, або текстове поле з Word в Excel; інакше кажучи, будь-який об'єкт, створений в одній програмі, копіюєте в іншу програму - ви використаєте OLE.

У контексті Lazarus variant може містити будь-яке число - ціле, або дійсне, рядок, логічне значення, час-дату, масив, той же OLE-об'єкт. При цьому компілятор сам визначає реальний тип варіанта на етапі виконання програми. Причому відразу ж виділяє під змінну цього типу пам'ять - якщо значення не зазначене, цієї змінної привласнюється нуль.

Давайте розглянемо наступний приклад:

```
var
  My: variant;
begin
  My:=10;   //варіант містить ціле число
  My:= 2.5;   //варіант містить уже дійсне число
  My:= 'Який то текст';   //варіант містить рядок
  My:= True;   //а тепер варіант містить логічне значення
```

Як видно із прикладу, ми привласнюємо однієї й того ж варіантної змінної значення різних типів. При цьому компілятор сам визначає найбільш підходящий за значенням тип, і переробляє варіантну змінну під нього. Наприклад, у рядку

```
  My:=10;
```

компілятор знайде найбільш підходящий під значення тип - Byte, і змінить змінну My під нього. Таким чином, програміст може не турбується про типи - за нього цю роботу виконає компілятор. У вищеописаному прикладі змінна My змінила свій тип 4 рази!

Однак за все доводиться платити. Змінна типу variant займає набагато більше пам'яті, ніж змінна будь-якого іншого типу (за винятком строкової змінної, у яку можна помістити обидві «Війна й мир»). Тому, використайте тип variant тільки тоді, коли це дійсно необхідно, наприклад, коли на етапі програмування ви не можете

заздалегідь визначити, дані якого типу буде змінна. В інших випадках використання типу variant невиправдано.

Зі змінними типу variant можна робити різні дії, наприклад, складати. При цьому різні типи даних будуть перетворювати під потрібний тип, якщо це можливо. Давайте спробуємо роботу зі змінної типу variant на практиці. Відкрийте **Lazarus** з новим проєктом. Не змінюючи імен за замовчуванням, відразу збережіть проєкт у папку **24-02**. На форму встановіть тільки одну кнопку й згенеруйте для неї подію OnClick. Його код наступний:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  v1, v2, v3: variant;
begin
  v1:= ‘5’;  //привласнили символ
  v2:= ‘10’;  //тепер рядок
  v3:= 20;  //тепер ціле число
  v1:= v1 + v2 + v3;  //тепер усі склали

  ShowMessage(IntToStr(v1));
end;
```

Не квапитеся запускати цю програму, спочатку спробуйте відповісти: яке число в підсумку виведе повідомлення ShowMessage? На перший погляд, результат буде 35 (5 + 10 + 20). Однак не поспішаєте з висновками, подумайте. Ми до символу ‘5’ додали рядок ‘10’. Оскільки всі однотипні операції виробляються праворуч, то в результаті ми одержимо рядок ‘510’. А потім ми до цього рядка додаємо ціле число 20. Варіантна змінна, котра спочатку мала символьний тип, потім строковий, одержала цілочисельний тип! А якщо до числа 510 додати 20, то скільки вийде? Звичайно, 530! Збережіть проєкт, запустіть його на виконання й переконайтеся в цьому самі.

3.Сітка рядків TStringGrid

Наостанок познайомимося із ще одним корисним інструментом - сіткою рядків TStringGrid. Це сітка, схожа на сітку в електронній таблиці MS Excel, вона дозволяє вводити й редагувати табличну інформацію у вигляді рядків і колонок.

Завантажите **Lazarus** з новим проєктом. Форму перейменуйте в **fMain**, проєкт збережете в папку **24-03** під ім’ям **Setka**, а модулю

форми дайте ім’я Main. Щоб ваша форма відповідала моєї, Встановіть ширину width = 625, а висоту height = 195. В Caption напишіть «Редагування сітки TStringGrid».

Тепер перейдіть на вкладку **Additional**, знайдіть і встановіть на форму компонентів TStringGrid. Поруч із ним ви можете побачити ще одну сітку, TDrawGrid. Різниця між ними в тому, що сітка TStringGrid самостійно промальовує дані в осередках, а в TDrawGrid це потрібно робити вручну. Тому що TStringGrid взучніша, її й будемо вивчати. У сітці встановіть ширину 615, а висоту 120 пікселів.

Нижче встановіть одну просту кнопку **TButton**, в Caption якої напишіть «Додати». У результаті, у вас повинна вийти от така форма:

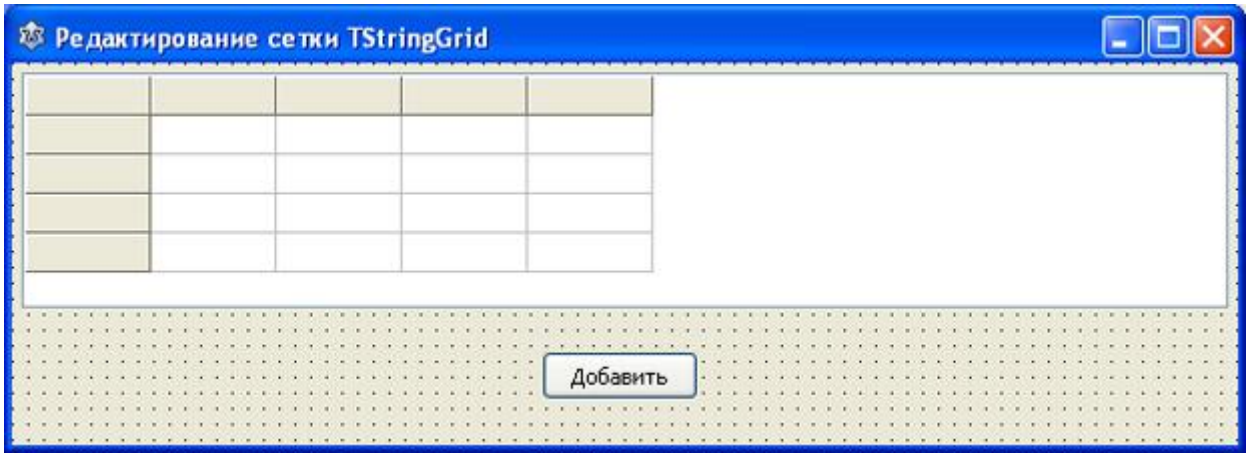


Рис. 24.3. Форма проєкту

Перш, ніж приступи до кодування, виділіть на формі сітку й подивіться в **Інспекторі об’єктів** на її властивості. Познайомимося з найбільш значимими з них.

BorderStyle	- стиль обрамлення. Може мати тільки два значення - з обрамленням і без нього.
ColCount	- кількість стовпчиків у сітці. За замовчуванням їх 5. При необхідності, це значення можна зминювати програмно, збільшуючи або зменшуючи кількість стовпчиків.
DefaultColWidth	- ширина стовпчиків за замовчуванням. На жаль, всім стовпчикам установлюється однакова ширина, хоча, як правило, різні стовпчики повинні мати різну ширину. Під час роботи програми ширину кожного стовпчика можна буде змінити програмно.

DefaultDrawing	- промальовування даних за замовчуванням. Якщо коштує True, то компонент сам буде відображати уведені дані, інакше це прийде робити програмістові.
DefaultRowHeight	- висота рядків за замовчуванням. Установлено 20 пикселей, але цей розмир небагато великуватий, рекомендую змінити його на 18. Хоча це на аматора, і залежить від розміру шрифту.
FixedCols	- кількість фіксованих стовпчиків. Вони виділяються сірими кольорами, і завжди перші. Це властивість можна назвати заголовком рядків. Практично не буває необхідності робити більше одного такого стовпчика, а іноді фіксовані стовпчика й зовсім не потрібні. За замовчуванням встановлений один стовпчик.
FixedRows	- кількість фіксованих рядків. За замовчуванням теж одна, і працює так само, як і FixedCols. Як правило, цей рядок служить заголовком стовпчиків.
GridLineWidth	- товщина розділових ліній. Спробуйте поставити нуль - лінії зникнуть. Поверніть одиницю.
Options	- сама головна властивість компонента. Воно містить багато настроювань, які розкриються, якщо клацнути по плюсі ліворуч від назви властивості. Ці настроювання є вимикачами (True-дозволено, False-заборонене). Всі настроювання ми розглядати не будемо, їх багато, і більшість із них звичайно не використається, крім того, у довіднику Lazarus немає описів цих настроювань. Розглянемо основні:
	goAlwaysShowEditor - якщо True, те редагувати осередок можна відразу при виділенні. Якщо False, то для редагування потрібно нажати **<Enter>** або **<F2>**.
	goColMoving - чи можна мишею перемищати стовпчика на інше місце.

	goColSizing	- дозволено зминювати ширину стовпчика мишею.
	goFixedVertLine	- чи малювати вертикальні лінії у фіксованих осередків? За замовчуванням True.
	goDrawFocusSelect	- дозволено виділяти осередок, що перебуває у фокусі уведення.
	goEditing	- чи можна редагувати сітку? Тобто, уводити дані із клавіатури. За замовчуванням установлене False, тобто, редагувати не можна. Ігнорується, якщо включено елемент goRowSelect. Якщо ви хочете, щоб користувач миг уводити в сітку дані, а звичайно це потрібно, цей параметр варто встановити в True.
	goFixedVertLine	- промальовування вертикальних ліній у фіксованих осередків.
	goHorzLine	- промальовування горизонтальних ліній у нефіксованих осередків.
	goRangeSelect	- дозвіл виділяти кілька осередків. Не працює, якщо включено елемент goEdit.
	goRowMoving	- чи можна мишею перемищати рядка на інше місце.
	goRowSelect	- виділяється весь рядок. Якщо дорівнює False, то тільки один осередок.
	goRowSizing	- дозволено зминювати висоту рядка мишею.

	goTabs	- чи можна перемикатися на інші осередки за допомогою клавіші <**Tab**>.
	goThumbTracking	- чи дозволене промальовування даних в осередках при прокручуванні. Якщо ні, то дані будуть обновлені після прокручування.
	goVertLine	- промальовування вертикальних ліній у всіх інших (нефіксованих) осередків.
RowCount	- кількість рядків у сітці. За замовчуванням їх 5, як і колонок.	
Row	- закрита властивість, містить індекс поточної (виділеної) рядка. Рядок може бути виділена цілком, або в ній може бути виділений тільки один осередок. Оскільки властивість закрита, то в **Інспекторі об'єктів** ви його не знайдете, однак при кодуванні, програмно, ця властивість доступно. Ви можете довідатися індекс поточного рядка, але не зможете його змінити за допомогою властивості Row.	

Також сітка TStringGrid має пари корисних методів, з якими теж варто познайомитися.

DeleteRow - метод видаляє рядок, індекс якої вказується в параметрі. Наприклад:

```
StringGrid1.DeleteRow(StringGrid1.Row);  //видалити поточний рядок
StringGrid1.DeleteRow(0);  //видалити першу
StringGrid1.DeleteRow(StringGrid1.RowCount-1);  //видалити останню
```

SortColRow - метод виконує сортування даних у сітці як по рядках, так і по стовпчиках. Має два параметри:

```
SortColRow(IsColumn: Boolean;  index: Integer);
```

Якщо IsColumn має значення True, то сортування виробляється по стовпчику з індексом index. Якщо False, то по рядку з індексом index. Як правило, дані потрібно сортувати саме по стовпчиках, наприклад, по стовпчику із прізвищами.

Про що ще варто знати? Дані сітки втримуються в окремих осередках. До осередку можна звертатися по її індексі, як до елемента двомірного масиву, наприклад:

```
s:=  StringGrid1.Cells[1, 1];  //вважали  значення
StringGrid1.Cells[1, 1]:= ‘Нове значення’;  //записали значення
```

Як видно із прикладу, осередки зберігаються в списку осередків Cells, при звертанні до осередку першим індексом є номер стовпчика, другим - рядка. Індексація починається з нуля, тому самий верхній лівий осередок буде мати індекс [0, 0]. Якщо в сітці є фіксовані рядки й(або) стовпчика, саме вони будуть мати нульовий індекс. Тобто, якщо потрібно вказати у фіксованому рядку заголовок третього стовпчика, робити це треба так:

```
StringGrid1.Cells[2, 0]:= ‘Заголовок’;
```

Змінювати ширину окремих стовпчиків потрібно програмно, використовуючи властивість ColWidth:

```
ColWidth[<індекс_стовпчика>]
```

Наприклад, так:

```
StringGrid1.ColWidths[0]:= 120;
```

Таким чином, можна буде зробити стовпчики різної ширини, чого не можна домогтися при проєктуванні форми, в **Інспекторі об’єктів**.

Ще в осередках можна використати маски введення для рядків і стовпчиків. Для цього потрібно згенерувати подію сітки OnGetEditMask, у якому й прописати маску. Як це робиться, буде видно з коду приклада нижче.

Доробимо сітку. Висоту рядків (властивість DefaultRowHeight) встановіть в 18 пикселей. Розгорніть властивість Options і встановіть в True параметр goEditing, щоб користувач міг редагувати дані сітки. Кількість фіксованих рядків і стовпчиків залишимо за замовчуванням - 1. Це будуть заголовки стовпців і рядків. Їх потрібно заповнити, для цього виділіть форму й згенеруйте для неї подію OnCreate. Код події такий:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  //заповнюємо заголовки рядків:
  StringGrid1.Cells[0, 1]:= ‘Іванов И.И.’;
  StringGrid1.Cells[0, 2]:= ‘Петров П.П.’;
  StringGrid1.Cells[0, 3]:= ‘Миколаїв Н.Н.’;
  StringGrid1.Cells[0, 4]:= ‘Бонд Дж.’;
  //заповнюємо заголовки стовпчиків:
  StringGrid1.Cells[1, 0]:= ‘Рік народження’;
  StringGrid1.Cells[2, 0]:= ‘Місце народження’;
  StringGrid1.Cells[3, 0]:= ‘Прописка’;
  StringGrid1.Cells[4, 0]:= ‘Родиний стан’;
  //миняємо ширину стовпчиків:
  StringGrid1.ColWidths[0]:= 120;
  StringGrid1.ColWidths[1]:= 90;
  StringGrid1.ColWidths[2]:= 150;
  StringGrid1.ColWidths[3]:= 100;
  StringGrid1.ColWidths[4]:= 120;
end;
```

Код, думаю, простий і зрозумілий. Як тільки програма завантажиться, сітка буде настроєна. Тепер для стовпчика «**Рік народження**» створимо маску введення, щоб користувачеві залишалося тільки вписати цифри. Виділіть сітку, в **Інспекторі об’єктів** перейдіть на вкладку **Події** й згенеруйте подію OnGetEditMask. Код події наступний:

```
procedure TForm1.StringGrid1GetEditMask(Sender: TObject; ACol, ARow: Integer;
  var Value: string);
begin
  if ACol=1 then Value:= ‘99.99.9999 р.’;
end;
```

Як бачите, у події стають доступними такі змінні, як ACol і ARow, у яких перебуває індекс відповідно, стовпчики й рядки. У змінну Value варто прописати текст маски, ми це робимо, тільки якщо поточною є другий стовпчик (індексація з нуля):

```
  if ACol=1 then Value:= ‘99.99.9999 р.’;
```

Тепер запрограмуємо кнопку, давши користувачеві можливість додавати нові прізвища. Згенеруйте для кнопки подія OnClick, код події наступний:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  s: string;
begin
  if InputQuery('Вкажіть нове ім'я',
            'Впишіть прізвище, ім'я та по батькові у форматі «Прізвище І.Б.»:',
            s) then begin
    StringGrid1.RowCount:= StringGrid1.RowCount + 1;
    StringGrid1.Cells[0, StringGrid1.RowCount-1]:= s;
  end;
end;
```

За допомогою функції-запиту InputQuerry ми одержуємо в змінну s прізвище, ім'я та по батькові. Потім додаємо ще один рядок:

```
StringGrid1.RowCount:= StringGrid1.RowCount + 1;
```

На закінчення, в останній рядок першого (фіксованого) стовпця ми вписуємо ці нові ФИО:

```
StringGrid1.Cells[0, StringGrid1.RowCount-1]:= s;
```

Причому робиться це тільки в тому випадку, якщо користувач їх вказав, інакше нічого не відбувається. Що ж стосується редагування інших даних, то користувач буде робити це прямо в програми, в осередках сітки.

От і все. Збережете проєкт, запустіть його на виконання й випробуйте сітку в справі:

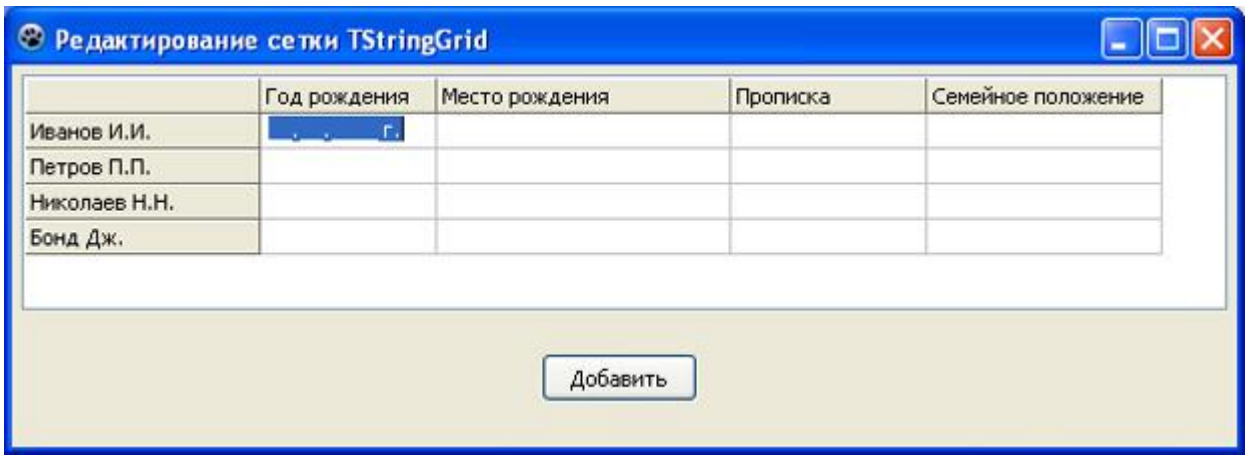


Рис. 24.4. Додаток у роботі

Тема 25. Телефонний довідник

У процесі роботи докладно розглядається створення програми - телефонного довідника.

У даній роботі нам має бути написаний додаток - телефонний довідник. Це додаток повиннен мати всі необхідні для нашого завдання якості:

- виводити контактні дані у вигляді таблиці;
- зберігати контакти у файл і зчитувати їх відтіля;
- додавати новий або редагувати (зминювати) існуючий контакт;
- видаляти обраний зі списку контакт;
- сортувати список контактів за алфавітом.

Для роботи з контактами створимо тип даних запис, будемо використати змінну цього типу й типізований файл, у якому дані будуть зберігатися у двійковому виді. Сами контакти будемо виводити в сітку TStringGrid, для додавання й редагування записів створимо окреме вікно - сітка дозволить тільки переглядати й вибирати контакти. Приступимося?

Реалізація

Для початку відкрийте **Lazarus** з новим проєктом. Форму, як звичайно, називаємо **fMain**, зберігаємо проєкт у папку **25-01** під ім'ям **telephones**, модулю головної форми привласнюємо ім'я Main. Підемо далі. Швидше за все, ви будете ділитися цією програмою із друзями,

тому має сенс відразу ж відключити від проєкту вставку налаштування. Якщо ви памятаєте, потрібно вибрати команду «**Проєкт -> Параметри проєкту**», у розділі «**Параметри компілятора**» перейти на підрозділ «**Компонування**» і забрати прапорець «**Генерувати отладочную інформацію для GDB**». Це дозволить нашій програми відразу «схуднути» - від 15 мегабайт готового продукту, до менш чим 2-х мегабайт.

Тепер займемося формою. У властивості Caption напишіть «Телефонний довідник», в BorderStyle виберіть bsDialog (нам не потрібно, щоб користувач миняв розміри вікна), а у властивості Position - poDesktopCenter. Щоб ваш додаток у точності відповідало моєму, виберіть ширину форми Width 700 пикселей, а висоту Height 400 пікселів.

Тепер нам на формі буде потрібна проста панель інструментів. Встановіть звичайну панель **TPanel**, очистіть її властивість Caption, у властивості Align виберіть alTop, щоб панель зайняла весь верх форми, і у властивості Height встановіть значення 27 - нам не потрібна занадто висока панель інструментів, вузька буде виглядати краще.

На панель нам потрібно встановити чотири кнопки **TSpeedButton** із вкладки **Additional Палітри компонентів**. Кнопки **TSpeedButton** нам потрібні, щоб на них не переходив фокус введення, у цьому випадку, завжди активною буде тільки сітка. Привласніть властивості Name кнопок відповідно, імена bAdd, bEdit, bDel і bSort. Перша «b» в іменах означає, що це кнопка - button. У властивостях Caption цих кнопок напишіть, відповідно, «Додати контакт», «Редагувати контакт», «Видалити контакт» і «Сортувати список» (без лапок, розуміє). Картинок на кнопки ми ставити не будемо, обійдемося тільки пояснювальним текстом. Властивості Top всіх кнопок привласніть значення 2, щоб вони майже пригорнулися до верхньої границі панелі, властивості Width привласніть 150, щоб весь текст містився на кнопках без проблем. Перемістіть їх відносно один одного так, щоб від лівого краю панелі й один від одного їх відокремлювало теж приблизно по 2 пікселі.

Тепер сітка. На вільне місце форми, нижче панелі, встановіть сітку TStringGrid. Оскільки нам часто прийдеться звертатися до неї в процесі кодування, вкоротимо ім’я у властивості Name до SG. У властивості Align сітки виберіть alClient, щоб сітка зайняла все місце, що залишилося. Тепер потрібно змінити деякі інші налаштування. За замовчуванням, у сітці встановлено по п’ять стовпчиків ColCount, і по

п’ять рядків RowCount. Стовпців нам потрібно тільки три: імена контакту, його телефон і примітка, який це телефон (мобільний, домашній, службовий, сусідський). Тому зменшимо кількість стовпчиків ColCount до 3. А рядок нам потрібен й зовсім одн, він буде служити заголовком стовпчиків, і він буде фіксованим. Зменшуйте RowCount до 1.

Тепер про фіксовані рядки й стовпці. Фіксований рядок FixedRows нам потрібен тільки один, це значення за замовчуванням. А фіксовані стовпці FixedCols нам і зовсім не потрібні, отут поставте нуль.

Інші параметри сітки можна залишити без змін. Якщо ви все зробили правильно, у вас у результаті повинна вийти така форма:

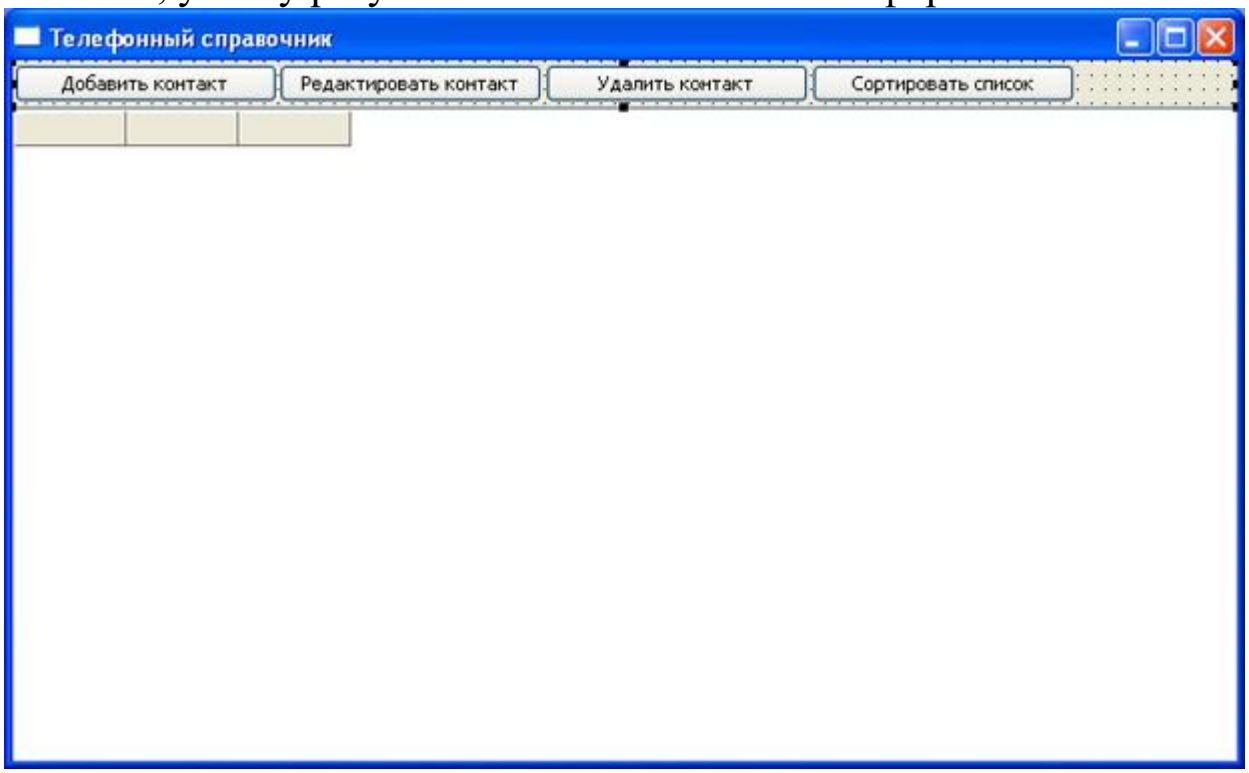


Рис. 25.1. Головна форма проєкту

Підемо далі. Перш, ніж приступати до самого кодування, давайте зробимо ще одну форму - вікно редакторів контактів. Командою **Файл -> Створити форму** створіть нову форму. Привласніть їй ім’я **fEdit**, натисніть кнопку «**Зберегти всі**» і дайте модулю форми ім’я Edit.

У властивості Caption напишіть «Редактор контакту». Чому не «контактів»? Щораз ми будемо завантажувати в це вікно по одному контакту, тому й використана однина. У властивості BorderStyle також виберіть bsDialog, у властивості Position - poMainFormCenter, щоб вікно з’являлося по центрі головної форми, де б вона не перебувала. Розміри форми: ширина width = 400, висота height = 225 пікселів.

Тепер розмістимо на формі компонента. Встановіть мітку TLabel, в Caption якої напишіть «Вкажіть ім’я контакту:».

Нижче встановіть рядок TEdit. Перейменуйте його в eName, ширину зробіть 380. Очистіть властивість Text.

Нижче ще одну мітку, з текстом в Caption «Вкажіть телефон:».

Нижче встановіть ще один TEdit. Перейменуйте його в eTelephone, ширину зробіть 210. Очистіть властивість Text.

Нижче ще одну мітку, з текстом в Caption «Виберіть тип телефону:».

Нижче нам буде потрібно список вибору TComboBox. Ми ж не хочемо, щоб користувачеві щораз самому доводилося писати «мобільний» або «службовий»? Перейменуйте TComboBox в CBNote, відкрийте редактор властивості Items і там наберіть наступні рядки:

Мобільний
Домашній
Службовий
Сусідський

Якщо зможете придумати інші типи, можете дописати їх сюди.

У властивості ItemIndex компонента встановіть 0, щоб у списку з’явився перший рядок «**Мобільний**». Ширину зробіть 210.

Ще нижче встановіть на одному рівні кнопки **TBitBtn**. Першу перейменуйте в bSave, у властивості Kind виберіть bkOK, щоб на кнопці з’явилася картинка - зелена галочка, переконаєтеся, що властивість ModalResult кнопки перейшло в mrOk. У властивості Caption напишіть «Зберегти контакт». Щоб текст нормально вміщувався, збільште її ширину до 160 пікселів.

Другу кнопку перейменуйте в bCancel, у властивості Kind виберіть bkCancel, а в Caption напишіть «Не зберігати». Властивість ModalResult кнопки повинна перейти в mrCancel. У результаті, у вас повинна вийти от така форма:

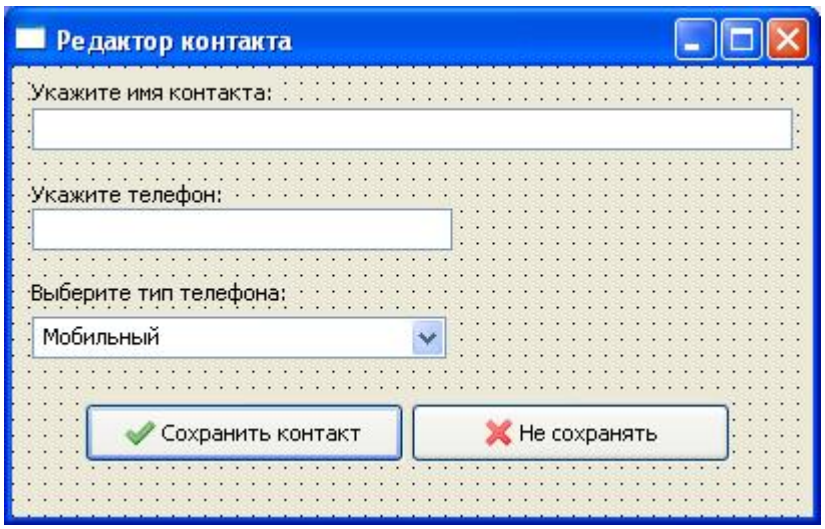


Рис. 25.2. Форма проєкту fEdit

Щоб більше не повертатися до цієї форми, давайте відразу введемо весь необхідний код форми, тим більше що його зовсім мало. Для кнопок нам взагалі не потрібно писати код. Коли ми змінили їх властивості Kind, встановивши там bkOK і bkCancel, відразу змінилася і їхня властивість ModalResult, встановившись, відповідно, в mrOK і mrCancel. І тепер натискання на кожну із цих кнопок приведе до закриття форми, і до того, що в ModalResult форми потрапить значення ModalResult натиснутої кнопки. Тобто, ми зможемо довідатися, якою кнопкою користувач закрив вікно.

Але нам потрібно все-таки зробити деяку підготовчу роботу. Перш, ніж показувати форму, нам потрібно буде перевести фокус введення в перший TEdit, адже від колишньої роботи з формою виділеним міг залишитися й інший компонент - кнопка або список вибору, наприклад. А це буде незручно для користувача, йому доведеться зайвий раз кликати мишею, виділяючи перший рядок.

Виділіть форму, клацнувши по вільній ділянці, в Інспекторі об’єктів перейдіть на вкладку **Події** й згенеруйте подію форми OnShow - це подія виникає щораз перед показом форми - ідеальне місце для підготовчої роботи. Код події зовсім простий:

```
procedure TfEdit.FormShow(Sender: TObject);
begin
  eName.SetFocus;
end;
```

І з редактором контактів це все! Вся інша робота буде відбуватися в головній формі.

У редакторі форм перейдіть на вкладку **Main** - наша головна форма. Насамперед, у розділі uses головної форми, через кому, додайте модуль edit - це наш редактор контактів. Якщо ми не додамо його в розділ uses, то не зможемо з ним працювати з головної форми.

Далі, **перед** розділом глобальних змінних, там, де оголошена змінна форми fMain, створимо ще один розділ type, у якому опишемо запис:

```
type
  Contacts = record
    Name: string[100];
    Telephon: string[20];
    Note: string[20];
  end; //record
```

У цьому записі три поля. Перше поле Name призначене для імені контакту. Ви можете помітити, що воно обмежено розміром 100 символів. Як уже говорилося в темі № 24, у записі можна вказувати необмежений тип string, але тоді створити типізований файл цього типу не вийде. Тому в нас всі рядки мають фіксований розмір. У поле Name користувач буде вписувати імена контактів, наприклад:

Анатолій Васильович, директор
Дмитрик, друг
Леночка, секретар

Може здатися, що 100 символів для імені контакту - це перебір. Однак згадаєте, що для тексту в нас використається кодування UTF-8, а в ній кожний символ кирилиці займає 2 байти. Тобто, якщо писати імена по-українськи, реально можна вписати тільки 50 символів! А прізвища й імена бувають дуже довгими, але 50 символів має вистачити. Для телефону ми вказали 20 символів. Це звичайно, цифри, дужки й знаки тире:

111-11-11
3-(222)-111-11-11

На кожний із цих символів потрібно тільки 1 байт, так що 20 символів вистачить позаочі для будь-якого телефону.

Ну й, нарешті, поле Note, у якому буде одне зі значень:

Мобільний
Домашній
Службовий

Сусідський

Ці символи теж займають по 2 байти, саме довге слово має 9 символів - 18 байт. А ми вказали 20, так що повинне вистачити.

Далі, у розділі глобальних змінних, після опису самої форми, опишемо глобальну змінну adres:

```
var
  fMain: TfMain;
  adres: string; //адреса, звідки запущена програма
```

Ця змінна знадобиться нам пізніше, у неї ми запишемо адресу, звідки була запущена програма, щоб потім по цій адресі відшукати файл із контактами. Раніше ми говорили, що якщо не вказувати адресу, то комп'ютер буде шукати файл у поточній папці, тобто - у папці із програмою. Однак так буває не завжди. Якщо ви паралельно працюєте з декількома іншими програмами, і всі вони раз у раз звертаються до файлів, те поточна адреса може змінитися. От чому завжди краще підстрахуватися, і вказати дійсна адреса - це гарантує вашу програму від будь-яких помилок.

Далі, «навчимо» програму приймати нові дані й виводити їх у сітку. Згенеруйте подію OnClick для кнопки «**Додати контакт**». Код події такий:

```
procedure TfMain.bAddClick(Sender: TObject);
begin
  //очищаємо поля, якщо там щось є:
  fEdit.eName.Text:= ‘‘;
  fEdit.eTelephone.Text:= ‘‘;
  //встановлюємо ModalResult редактори в mrNone:
  fEdit.ModalResult:= mrNone;
  //тепер виводимо форму:
  fEdit.ShowModal;
  //якщо користувач нічого не ввів - виходимо:
  if (fEdit.eName.Text= ‘‘) or (fEdit.eTelephone.Text= ‘‘) then exit;
  //якщо користувач не нажав «Зберегти» - виходимо:
  if fEdit.ModalResult <> mrOk then exit;
  //інакше додаємо в сітку рядок, і заповнюємо її:
  SG.RowCount:= SG.RowCount + 1;
  SG.Cells[0, SG.RowCount-1]:= fEdit.eName.Text;
  SG.Cells[1, SG.RowCount-1]:= fEdit.eTelephone.Text;
```

```
  SG.Cells[2, SG.RowCount-1]:= fEdit.CBNote.Text;
end;
```

Давайте розбиратися з кодом. Насамперед, у редакторі контактів fEdit ми очистили текст із рядків з ім'ям і номером телефону. Адже якщо користувач уже додавав або редагував контакти, у цих рядках залишаться записи з минулого сеансу. Потім властивості ModalResult форми ми привласнюємо значення mrNone - немає результату. Робимо це потім, щоб потім точно знати, яким образом користувач закрив вікно редактора контактів. Якщо кнопкою «**Зберегти контакт**», то в цьому випадку ModalResult форми буде mrOk, якщо кнопкою «**Не зберігати**», ModalResult форми буде mrCancel. Якщо ж він просто закрив вікно хрестиком у правому верхньому куті вікна або клавішами **<Alt+F4>**, то в цьому випадку ModalResult форми так і залишиться mrNone.

Далі, ми виводимо вікно редактора контактів, як модальне.

До наступного рядка коду програма добереться, коли користувач введе (або не введе) новий контакт і закриє вікно редактора. Перш, ніж щось додавати в сітку, ми робимо перевірку - а чи ввів користувач ім'я нового контакту, і телефон? Адже якщо він нічого не ввів, то й зберігати нема чого. І якщо він увів один тільки телефон, без імені, або ім'я без телефону - який зміст зберігати такий контакт? Тому, якщо немає імені або телефону, ми просто виходимо із процедури:

```
  if (fEdit.eName.Text= ‘‘) or (fEdit.eTelephone.Text= ‘‘) then exit;
```

Однак може трапитися, що користувач увів і ім'я, і телефон, але потім передумав зберігати контакт і нажав кнопку «**Не зберігати**», або просто закрив вікно редактора. Тому нам потрібно зробити ще одну перевірку:

```
  if fEdit.ModalResult <> mrOk then exit;
```

Тут ми дивимося значення властивості ModalResult форми. Воно зможе стати mrOK тільки в одному випадку - якщо користувач нажав кнопку «**Зберегти контакт**». І якщо це не так, то виходимо із процедури, нічого не зберігаючи.

Якщо ми не вийшли із процедури в попередніх двох перевірках, то це означає, що

1. нам є, що зберігати
2. користувач нажав «**Зберегти контакт**»

І якщо це так, виходить, додаємо новий рядок у сітку SG і заповнюємо три її стовпчики значеннями з редактора контактів. Додаткових перевірок на існування тексту примітки ми не робили, тому що там у кожному разі буде зазначений один з типів телефонів:

```
//інакше додаємо в сітку рядок, і заповнюємо її:
SG.RowCount:= SG.RowCount + 1;
SG.Cells[0, SG.RowCount-1]:= fEdit.eName.Text;
SG.Cells[1, SG.RowCount-1]:= fEdit.eTelephone.Text;
SG.Cells[2, SG.RowCount-1]:= fEdit.CBNote.Text;
```

Збережіть проєкт і запустіть його на виконання. Якщо ви все зробили вірно, програма дозволить вам додавати нові контакти. Не звертайте уваги на недопрацьований вид сітки - вузькі стовпці без заголовків - це ми виправимо трохи пізніше.

Але цього мало, хотілося б ще й зберігати ці контакти у файл. Для цього нам найкраще підійде подія форми OnClose, що буде виконуватися тільки один раз за сеанс роботи програми - перед її закриттям. Форма вся закрита компонентами, але по минулих темах ви повинні пам’ятати, що форму можна виділити у верхнім віконці **Інспектора об’єктів**, де компоненти зазначені у вигляді дерева, у самий вершині якого форма **fMain**. Ще можна виділити компонент верхнього над формою рівнів - панель або сітку, і нажати **<Esc>**. При цьому виділення перейде на більше низькі рівні - на форму.

Ітак, виділіть форму, в **Інспекторі об’єктів** перейдіть на вкладку **Події** й згенеруйте подію OnClose. Код події такий:

```
procedure TfMain.FormClose(Sender: TObject; var CloseAction: TCloseAction);
  var
    MyCont: Contacts; //для чергового запису
    f: file of Contacts; //файл даних
    i: integer; //лічильник циклу
  begin
    //якщо рядка даних порожні, просто виходимо:
    if SG.RowCount = 1 then exit;
    //інакше відкриваємо файл для запису:
    try
      AssignFile(f, adres + ‘telephones.dat’);
      Rewrite(f);
      //тепер цикл - від першої до останнього запису сітки:
```

```
    for i:= 1 to SG.RowCount-1 do begin
      //одержуємо дані поточного запису:
      MyCont.Name:= SG.Cells[0, i];
      MyCont.Telephon:= SG.Cells[1, i];
      MyCont.Note:= SG.Cells[2, i];
      //записуємо їх:
      Write(f, MyCont);
    end;
  finally
    CloseFile(f);
  end;
end;
```

Як бачите, ми оголосили змінну типу нашого запису Contacts, і типізований файл із таким же типом. Ще ми оголосили цілочисельну змінну, котра знадобиться нам для організації циклу.

Спочатку ми перевіряємо - є чи в сітках дані? Якщо SG.RowCount = 1, то це означає, що в сітці є тільки один рядок - фіксований, та, де в нас перебувають заголовки стовпчиків (а менше 1 рядка й бути не може, користувач не зможе видалити фіксований рядок). Інакше кажучи, даних немає. У цьому випадку ми просто виходимо із процедури.

Якщо програма пішла далі, виходить, дані в сітці є, правда? Тоді ми починаємо роботу зі збереження даних у файл. Цю роботу ми поміщаємо в блок try-finally-end, роботу з файлами (у тому числі типізованими) ми з вами неодноразово розглядали, так що докладно зупинятися отут не буду. Але пари зауважень зробити все-таки потрібно. По-перше, при асоціації з файлом ми вказали не тільки ім’я, але й адреса файлу, що повинен перебувати в змінної adres:

```
AssignFile(f, adres + ‘telephones.dat’);
```

Однак цю змінну ми оголосили, але нічого туди поки не внесли. Не звертайте на це уваги - ми зробимо це в наступному коді. Поки вважайте, що адреса там є, однак не поспішаєте запускати програму на виконання, точніше, не намагайтеся закрити цю програму, якщо там є дані.

По-друге, запис даних у файл ми організували у вигляді циклу:

```
    for i:= 1 to SG.RowCount-1 do begin
```

Тобто, ми робимо цю роботу від рядка з індексом 1 (нульовий індекс у фіксованого рядка, він нам для обробки даних не потрібен), до рядка з індексом SG.RowCount-1. Так, якщо в сітці в нас 10 рядків, то індекс останнього рядка = 9, а RowCount поверне 10, тому ми віднімаємо одиницю.

Всередині циклу ми зчитуємо в змінну типу запис дані із сітки, з поточного рядка даних:

```
MyCont.Name:= SG.Cells[0, i];
```

Коли розглядати всі значення, ми записуємо цю змінну в типізований файл. Оскільки типи змінної й файлу збігаються, то запис буде успішним, а покажчик зміститься в кінець файлу. Потім цикл продовжиться. При кожному новому кроці циклу значення і буде збільшуватися на одиницю, і код буде посилатися на новий наступний рядок. Записи будуть записуватися у файл один за одним, до самого кінця. У результаті в нас вийде щось схоже до бази даних.

Але щоб від нашої бази була користь, програма при завантаженні повинна ці дані зчитувати назад у сітку. Зручніше за все, як уже говорилося раніше, підготовчу роботу помищати в подію форми OnCreate. Згенеруйте цю подію, код буде таким:

```
procedure TfMain.FormCreate(Sender: TObject);
var
  MyCont: Contacts; //для чергового запису
  f: file of Contacts; //файл даних
  i: integer; //лічильник циклу
begin
  //спочатку одержимо адресу програми:
  adres:= ExtractFilePath(ParamStr(0));
  //настроїмо сітку:
  SG.Cells[0, 0]:= ‘Ім’я’;
  SG.Cells[1, 0]:= ‘Телефон’;
  SG.Cells[2, 0]:= ‘Примітка’;
  SG.ColWidths[0]:= 365;
  SG.ColWidths[1]:= 150;
  SG.ColWidths[2]:= 150;
  //якщо файлу даних ні, просто виходимо:
  if not FileExists(adres + ‘telephones.dat’) then exit;
  //інакше файл є, відкриваємо його для читання й
  //зчитуємо дані в сітку:
```

```
  try
    AssignFile(f, adres + 'telephones.dat');
    Reset(f);
    //тепер цикл - від першої до останнього запису сітки:
    while not Eof(f) do begin
      //зчитуємо новий запис:
      Read(f, MyCont);
      //додаємо в сітку новий рядок, і заповнюємо її:
        SG.RowCount:= SG.RowCount + 1;
        SG.Cells[0, SG.RowCount-1]:= MyCont.Name;
        SG.Cells[1, SG.RowCount-1]:= MyCont.Telephon;
        SG.Cells[2, SG.RowCount-1]:= MyCont.Note;
    end;
  finally
    CloseFile(f);
  end;
end;

Тут ми, насамперед, одержуємо адресу програми,
у змінну adres:
  adres:= ExtractFilePath(ParamStr(0));
```

Як ви пам'ятаєте з теми № 21 про консольні додатки, у будь-якому проєкті доступний параметр ParamStr(0). У цьому параметрі зберігається адреса й ім'я що завантажує програма. Функція ExtractFilePath відтинає ім'я файлу, повертаючи тільки його адресу із завершальною «\», наприклад:

C:\Program Files\MyProg

Адреса, звідки запущена програма, ми її зберігаємо в змінну adres. Оскільки вона глобальна, то стає доступною у всьому модулі головної форми. Тому ми скористалися цією змінною для одержання адреси в події OnClose, у якому прописали код збереження даних у файл. Тому що подія OnCreate виконується першою, то адреса в змінній adres дійсно, уже буде. Так що тепер дані будуть зберігатися.

Далі ми набудовуємо сітку - вказуємо ширину стовпців, вписуємо заголовки стовпчиків. Як це робиться, ми розбирали в минулій темі.

Потім ми дивимося - є чи файл? Програма може запускатися вперше, або користувач міг видалити старий файл, у кожному із цих випадків ми виходимо, не роблячи подальших дій. Сітка однаково вже

налаштована, так що програма буде виглядати, як треба. Та й адресу програми ми вже помистили в змінну adres.

Але якщо файл є, то потрібно взяти з нього дані, і помістити їх у сітку. Робота з файлом вам знайома, так що описувати процес не буду. Помичу тільки, що в цьому випадку ми скористалися циклом

```
while not Eof(f) do begin
```

Поки не кінець файлу. При відкритті файлу його покажчик встановлюється в початок. Коли ми зчитуємо перший запис у змінну MyCont, то покажчик перемищається до наступного запису. Так, раз за разом, зчитуючи запис у змінну, а потім, копіюючи з неї дані в сітку, ми рухаємося до кінця файлу, обробляючи всі записи, які там є. Причому при кожному кроці циклу ми спочатку додаємо в сітку новий рядок:

```
SG.RowCount:= SG.RowCount + 1;
```

а потім у цей, останній рядок, ми записуємо зчитані з файлу дані.

Отже, на даному етапі, ми навчили програму додавати нові записи (подія OnClick кнопки bAdd), навчили її зберігати ці записи у файл (подія форми OnClose) і зчитувати їх з файлу (подія форми OnCreate). Залишилося навчити програму редагувати існуючі записи - раптом користувач щось вирішить виправити, наприклад, змінився телефон контакту, видалити контакти, що стали непотрібними, а також сортувати список за алфавітом. Адже користувач буде вносити контакти, як прийдеться, не один за одним, а потрібний запис набагато простіше знайти в сортованому списку.

Отже, згенеруйте подію OnClick для кнопки bEdit («**Редагувати контакт**»). Код:

```
procedure TfMain.bEditClick(Sender: TObject);
begin
  //якщо даних у сітці немає - просто виходимо:
  if SG.RowCount = 1 then exit;
  //інакше записуємо дані у форму редактори:
  fEdit.eName.Text:= SG.Cells[0, SG.Row];
  fEdit.eTelephone.Text:= SG.Cells[1, SG.Row];
  fEdit.CBNote.Text:= SG.Cells[2, SG.Row];
  //встановлюємо ModalResult редактори в mrNone:
  fEdit.ModalResult:= mrNone;
  //тепер виводимо форму:
```

```
  fEdit.ShowModal;
  //зберігаємо в сітку можливі зміни,
  //якщо користувач нажав «Зберегти»:
  if fEdit.ModalResult = mrOk then begin
    SG.Cells[0, SG.Row]:= fEdit.eName.Text;
    SG.Cells[1, SG.Row]:= fEdit.eTelephone.Text;
    SG.Cells[2, SG.Row]:= fEdit.CBNote.Text;
  end;
end;
```

Давайте розбиратися з кодом. Як звичайно, ми виходимо із процедури, якщо даних у сітці немає - редагувати адже нема чого. Інакше перед показом вікна редактори контактів ми проробляємо підготовчу роботу - заносимо в рядки TEdit цієї форми дані контакту - ім'я й телефон. Також властивості Text списку вибору TComboBox ми привласнюємо текст примітки. Все це дозволить представити користувачеві дані поточного запису. Ви знаєте, як довідатися індекс поточного рядка сітки? Цей індекс зберігатися у властивості Row:

```
  fEdit.eName.Text:= SG.Cells[0, SG.Row];
```

Далі, ModalResult форми редактора ми встановлюємо в mrNone, щоб потім можна було визначити, яким чином користувач закрив цей редактор.

Потім ми виводимо форму, як модальне вікно. На закінчення ми перевіряємо - чи нажав користувач кнопку «**Зберегти контакт**»? Якщо так, то ми перезаписуємо дані в сітці на новий варіант, з форми редактора.

Натиснувши на кнопку «**Редагувати контакт**», користувач зможе скористатися цим кодом. Однак у хороших програмах користувачеві часто надають різні інструменти для виконання одного завдання. Наприклад, гарним тоном вважається редагувати запис сітки при подвійному кліку по ній. Давайте, реалізуємо цю можливість, тим більше що це зовсім просто. Виділіть сітку. В **Інспекторі об'єктів** перейдіть на вкладку **Події**. Тільки не квапитеся генерувати нову подію, адже писати новий код нам не потрібно. У події OnDblClick натисніть кнопку вибору, і виберіть там подію bEditClick - тобто те, що ми тільки що описали:

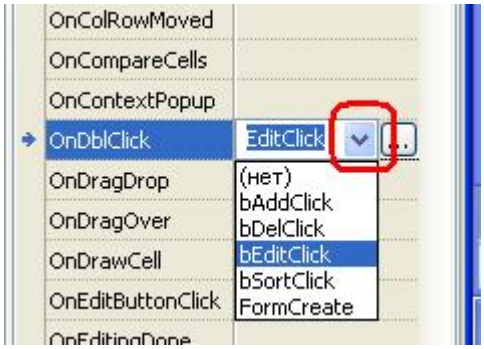


Рис. 25.3. Вибір уже існуючої події

Таким чином, і при натисканні кнопки «**Редагувати контакт**», і при подвійному кліку по запису сітки буде виконана та сама процедура.

Теперь навчимо програму видаляти контакти. Згенеруйте OnClick для кнопки «**Видалити контакт**»:

```
procedure TfMain.bDelClick(Sender: TObject);
begin
  //якщо даних немає - виходимо:
  if SG.RowCount = 1 then exit;
  //інакше виводимо запит на підтвердження:
  if MessageDlg('Потрібне підтвердження',
              'Ви дійсно хочете видалити контакт «' +
              SG.Cells[0, SG.Row] + '«?',
     mtConfirmation, [mbYes, mbNo, mbIgnore], 0) = mrYes then
       SG.DeleteRow(SG.Row);
end;
```

Цей код простіший, так? Знову ми виходимо із процедури, якщо даних немає, адже тоді й видаляти нема чого. Видалити запис просто, але потрібно чи це робити? Користувач міг натиснути на кнопку не подумавши, або випадково. Тому перед видаленням ми запросимо підтвердження:

```
  if MessageDlg('Потрібне підтвердження',
              'Ви дійсно хочете видалити контакт «' +
              SG.Cells[0, SG.Row] + '«?',
     mtConfirmation, [mbYes, mbNo, mbIgnore], 0) = mrYes then
       SG.DeleteRow(SG.Row);
```

Програма виведе подібне повідомлення:

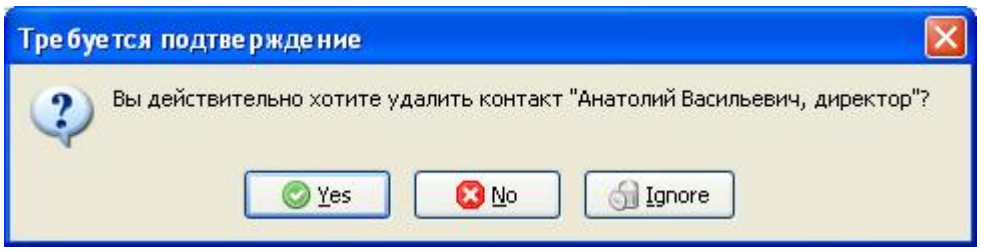


Рис. 25.4. Запит на підтвердження

І тільки якщо користувач підтвердить видалення контакту, він буде вилучений:

```
SG.DeleteRow(SG.Row);
```

Нам залишилося згенерувати подію OnClick для кнопки «**Сортувати список**». Його код:

```
procedure TfMain.bSortClick(Sender: TObject);
begin
  //якщо даних у сітці немає - просто виходимо:
  if SG.RowCount = 1 then exit;
  //інакше сортуємо список:
  SG.SortColRow(true, 0);
end;
```

Код досить простий, ми виконуємо сортування, якщо є дані, по першому стовпці (з іменами контактів):

```
  SG.SortColRow(true, 0);
```

Збережіть проєкт, запустіть його й спробуйте в роботі. Додаток має працювати, і виглядати приблизно так:

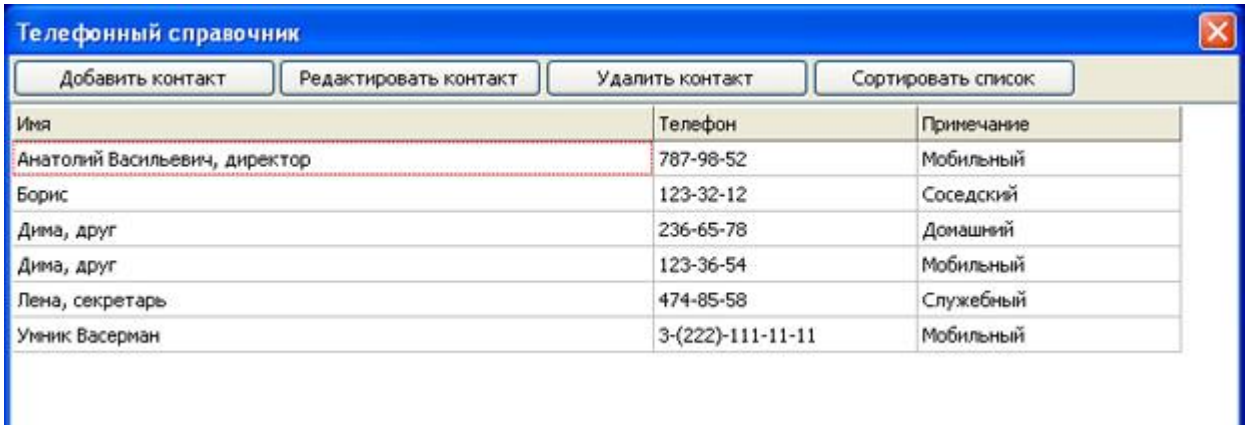


Рис. 25.5. Вікно працюючого додатка

Якщо є бажання, можете додатково розширити проєкт - самостійно зробити вікно «**Про програму**» до цього додатка, реалізувати механізм сортувань інакше, де користувач міг би вибрати спосіб сортування: за іменами контактів, за телефонами, або за типами телефонів.

Тема 26. DLL

1. Що таке DLL
2. Створення DLL
3. Динамічне зв’язування DLL

1.Що таке DLL

Сучасне програмування стало досить складним, уже написане кілометри рядків коду, і деякі його частини час від часу доводиться дублювати в різних програмах. Якби кожний програміст у кожній новій програмі заново писав весь необхідний код, мови програмування ніколи б не одержали такого стрімкого розвитку. Однак ми у своїх проєктах можемо використати інструменти, створені іншими програмістами - типи, процедури, функції... Ми вже неодноразово підключали до наших проєктів різні модулі й користувалися описаними в них засобами. І все було б добре, якби не одне АЛЕ...

Справа в тому, що при використанні модульного підходу, ми в момент компіляції проєкту впроваджуємо в нього весь той код, що був описаний у модулі. І наша програма від такого впровадження «товстіє», додає інший раз по трохи мегабайта. Здавалося б, дрібниці, сучасне апаратне забезпечення досить гарне розвинене, і ми можемо собі це дозволити. Але це все-таки тисячі рядків чужого коду, причому далеко не весь цей код задіяний у наших проєктах! Тепер уявіть, що ви пишете одну програму за іншою, впроваджуєте в неї ті самі модулі, у результаті ваші проєкти містять мегабайтів зовсім однакового коду. І добре б тільки це, але уявіть також, що ви одночасно завантажуєте й використаєте кілька програм (так здраствує багатозадачність!). А ці програми теж можуть мати у своєму складі безліч рядків дубльованого коду. От тепер у нас виходить дійсне марнотратство - ми дарна переводимо не тільки пам’ять жорсткого диска, де зберігаються програми, але й оперативну пам’ять, куди ми їх для використання завантажили!

Є ще один недолік модульного підходу - ми можемо використати чужий код, тільки якщо він написаний на такій же мові програмування, яким користуємося ми. Ну, в Lazarus ми можемо використати модулі Delphi, а також код на Асемблері. Але якщо код був написаний на C++ або Visual Basic, наприклад? Коротше кажучи, був потрібний новий підхід, що дозволяє використати чужий код, не впроваджуючи його безпосередньо в програму, і, крім того, «мовонезалежний». І таким підходом стала **DLL**.

DLL (англ., Dynamic Link Library - Бібліотеки, Що Підключають Динамично) - універсальний механізм впровадження в програму функцій і процедур, написаних іншими програмістами, і можливо, на іншій мові програмування. Впровадження відбувається не в момент компіляції проєкту, як у випадку модулів, а в момент завантаження готової програми в оперативну пам'ять.

Використання DLL дає нам ще одна дуже важлива перевага. Уявіть, що у вас десятки програм імпортують якийсь код. І от, цей код потрібно змінити, або доповнити, тому що ви перейшли на іншу ОС, на інші стандарти або просто тому, що керівництво зажадало від вас розширити функціональність ваших програм. Якби ви використали модулі, вам довелося б тяжко - потрібно було б переробляти не тільки цей модуль, але й перекомпілювати ваші програми. Але ви адже їх створювали не тиждень і не місяць, правда? Що, якщо частина з них ви писали колись на Delphi 7, іншу частину на Delphi XE3, а третю - уже на Lazarus? Робота була б та ще. Але з використанням DLL це завдання сильно спрощується. Вам потрібно переробити всього лише цю саму DLL-бібліотеку, а програми, що використають її, переробляти вже не потрібно!

Яким же образом використаються ці DLL? Фактично, DLL - це бібліотека тих інструментів, які можуть бути використані вами в різних проєктах. Причому з обліком того, що ви, може бути, будете використати різні мови програмування. Отже, першою справою, ви створюєте саму бібліотеку - файл *.dll. Створюється він, як окремий додаток, але після компіляції сам по собі працювати однаково не зможе, оскільки не має власної крапки входу, як виконуваний exe-файл.

Далі, ви створюєте додаток. Файл використовуваної динамічної бібліотеки ви розміщуєте в ту ж папку, що й проєкт. У випадку, якщо кілька ваших проєктів будуть використати цю бібліотеку, то її краще помістити в яку-небудь системну папку, наприклад,

C:\Windows\system32

Тоді всі ці програми зможуть довантажувати дану бібліотеку, не вказуючи її адреси - Windows переглядає всі свої системні папки й знає, де перебуває ця DLL. Функції й процедури, описані в DLL, наша програма буде сприймати, як свої власні. І коли під час виконання програми відбувається виклик таких функцій і процедур, то буде підвантажена відповідна DLL.

Під час компіляції проєкту код бібліотеки не буде впроваджений у програму. Це означає дві речі:

1. ваша програма не буде мати зайвого коду;
2. при поширенні програми між користувачами вам потрібно буде також подбати, щоб у них була встановлена й ця dll-бібліотека.

Коли користувач завантажує вашу програму, то також завантажується й використовуваний dll-файл, на це витрачається додатковий час. Зате бібліотека залишається в оперативній пам’яті навіть після завершення роботи її програми, що завантажила. І якщо потім буде завантажена інша програма, що використає ту ж DLL, вона вже не буде витрачати час на повторне завантаження цієї бібліотеки, а скористається вже завантаженої на згадку версією. Більше того, якщо будуть одночасно працювати кілька програм, що використають ту саму DLL, то вони не будуть завантажувати для себе по копії цієї бібліотеки, оскільки Windows гарантує, що одночасно може бути завантажена тільки одна копія динамічної бібліотеки. Так що перша із цих програм завантажить DLL в оперативну пам’ять, інші будуть використати її ж, не витрачаючи зайвого часу на завантаження. От чому такі програми, як MS Word або MS Excel повільно завантажуються в перший раз - вони звертаються до безлічі бібліотек, які також довантажуються на згадку. Зате коли ви завантажуєте MS Word удруге, він вантажиться набагато швидше, тому що потрібні бібліотеки вже в оперативній пам’яті! Відчули різницю між DLL і модулями?

До речі, динамічним бібліотекам прийнято давати розширення *.dll, однак це не завжди так. Ви можете дати такій бібліотеці будь-яке розширення, або навіть взагалі не вказувати його! Наприклад, драйвери пристроїв мають розширення *.drv, але це такі ж динамічні бібліотеки.

В DLL є один недолік: оскільки вони мовонезалежні, а також по деяких інших причинах, у цих бібліотеках не можна використати глобальні змінні, які будуть доступні в програми. Тому в DLL не

можна використати об’єкти (властивості об’єктів - це глобальні змінні). Так що сприймайте DLL, як набір процедур і функцій, що зручно довантажувати до різних програм.

У нашому курсі мається на увазі, що ми використаємо операційну систему сімейства Windows, однак механізм бібліотек, що підключають динамічно, існує у всіх сучасних операційних системах. В Windows така бібліотека являє собою файл *.dll, в Linux і Unix - *.so (Shared Object Library), а в MacOS X - *.dylib (Dynamic Shared Library). І всі ці бібліотеки можна створювати за допомогою Lazarus. З огляду на, що ми використаємо ОС Windows, ми будемо розглядати створення dll-файлів.

Створення DLL

Завантажуємо **Lazarus** і вибираємо команду «**Файл -> Створити…**»... Відкриється вікно, у якому нам потрібно буде вибрати «**Бібліотека**»:

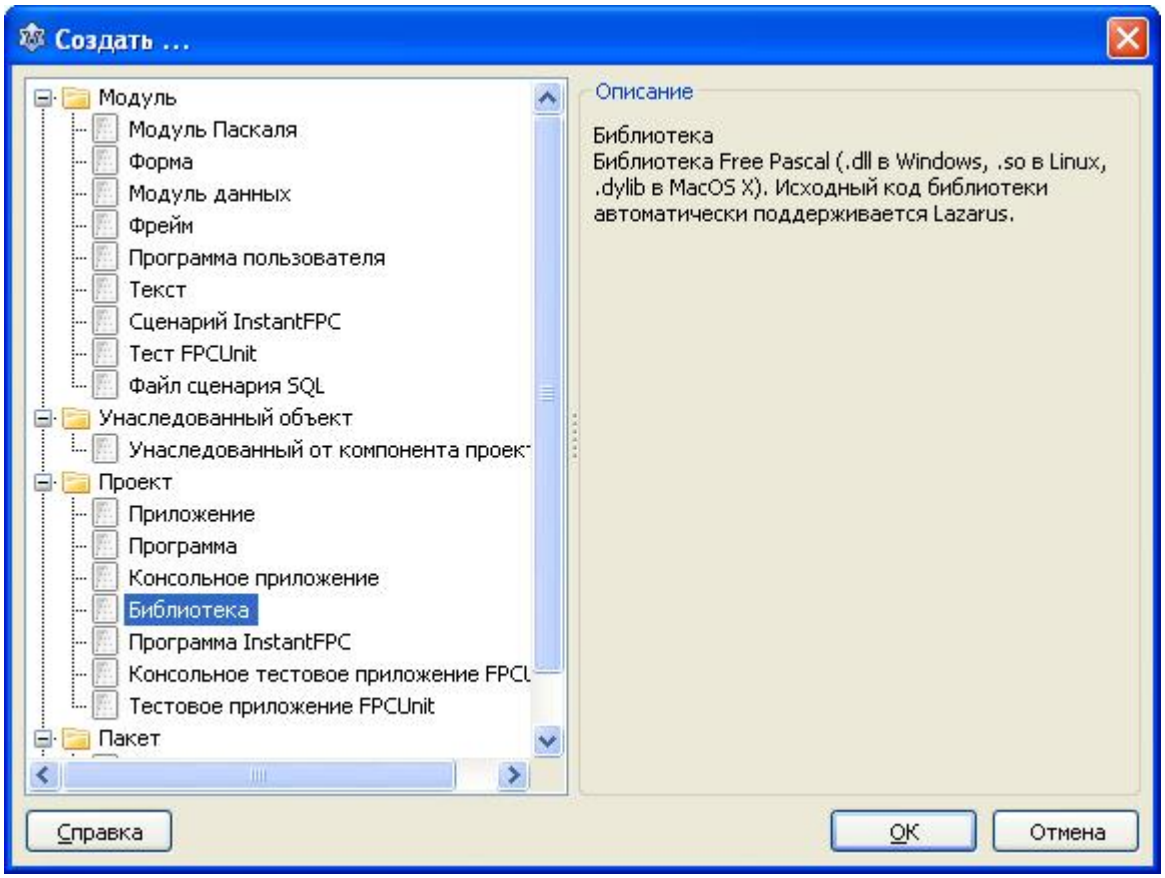


Рис. 26.1. Створення динамічної бібліотеки

У результаті буде створений проєкт, у якому буде перебувати наступний код:

```
library Project1;
```

```
{$mode objfpc}{$H+}

uses
  Classes
  { you can add units after this };

begin
end.
```

Натисніть кнопку «**Зберегти всі**», і збережете проєкт під ім'ям **MyFirstDLL** у папку **26-01**. Як бачите, перший рядок коду автоматично змінився на

```
library MyFirstDLL;
```

а в зазначеній папці з'явилося три файли: **MyFirstDLL.lpi**, **MyFirstDLL.lpr** і **MyFirstDLL.lps**. Ви можете відразу ж спробувати скомпілювати проєкт командою «**Запуск -> Компілювати**» або кнопками **<Ctrl+F9>** і в результаті одержите четвертий файл **MyFirstDLL.dll**. Це і є динамична бібліотека, що поки ще нічого не вмиє робити.

У розділі uses, після модуля Classes, через кому ми можемо додати й інші, необхідні нам модулі, однак не забувайте, що це буде збільшувати результуючий розмир бібліотеки. Тому потрібно включати тільки ті модулі, без яких дійсно, не обійтися. Зверніть увагу - після останнього модуля крапку з коми ставити не потрібно, вона коштує після коментарю, що повідомляє, що ви можете додавати модулі після цього, тобто Classes. Коментар можете залишити, а можете й видалити.

Весь основний код бібліотеки повинен перебувати над завершальними begin...end. Для прикладу навчимо наш модуль робити кілька корисних речей:

1. Переводити числа зі звичайних арабських у римські, і навпаки.
2. Робити просту шифрацію зазначеного рядка.
3. Повертати кількість днів до дня народження.

Оскільки нам доведеться мати справа з рядками, те варто мати на увазі от що: у динамічних бібліотеках рекомендують застосовувати рядковий тип PChar. Це пов'язане з тим, що вашою DLL можуть користуватися програмісти, що створюють додатки на інших

платформах. У середовищі Windows тип PChar уважається «рідним», всі WinAPI-функції використають цей тип. Усередині DLL-файлу можна використати будь-який строковий тип, але для передачі параметрів і для одержання результатів краще використати PChar.

Далі наводжу повний код нашої бібліотеки, після чого розберемо деякі моменти:

```
library MyFirstDLL;

{$mode objfpc}{$H+}

uses
  Classes, DateUtils, SysUtils
  { you can add units after this };

const
  R: array[1..13] of string[2] =
  (‘I’, ‘IV’, ‘V’, ‘IX’, ‘X’, ‘XL’, ‘L’, ‘XC’, ‘C’, ‘CD’, ‘D’, ‘CM’, ‘M’);
  A: array[1..13] of Integer =
  (1, 4, 5, 9, 10, 40, 50, 90, 100, 400, 500, 900, 1000);

{шифрація/дешифрація}
function Code(s: PChar; Key: integer): PChar; stdcall;
var
  i: integer;
  ss: string;
begin
  ss:= s;
  for i:= 1 to length(s) do
    ss[i]:= char(Ord(ss[i]) xor Key);
  Result:= PChar(ss);
end;

{днів до чергового дня народження}
function BeforeBirthday(Birthday:TDateTime): Integer; stdcall;
var
  d,m,y: word; //день, місяць і рік
  mydate: TDateTime;
begin
  mydate:= Birthday;
```

```
  //розберемо дату, змінимо рік на поточний і повернемо назад:
  DecodeDate(mydate, y, m, d);
  y:= YearOf(Date);
  mydate:= EncodeDate(y, m, d);
  //якщо ДР сьогодні:
  if mydate = Date then Result:= 0
  //якщо буде:
  else if mydate > Date then Result:= DaysBetween(Date, mydate)
  //якщо вже був:
  else begin
    //встановимо дату ДР на наступний рік і знову порахуємо:
    y:= y + 1;
    mydate:= EncodeDate(y, m, d);
    Result:= DaysBetween(Date, mydate);
  end; //else
end;

{Арабські в римські}
function ArToRom(N: integer): PChar; stdcall;
var
  i: integer;
  s: string;
begin
  s := ‘‘;
  i := 13;
  while N > 0 do begin
    while A[i] > N do Dec(i);
    s := s + R[i];
    Dec(N, A[i]);
  end;
  Result := PChar(s);
end;

{Римські в арабські}
function RomToAr(s: PChar): Integer; stdcall;
var
  i, p: Integer;
begin
  Result := 0;
```

```
    i := 13;
    p := 1;
    while p <= Length(s) do begin
      while Copy(s, p, Length(R[i])) <> R[i] do begin
        Dec(i);
        if i = 0 then Exit;
      end; //while 2
      Result := Result + A[i];
      p := p + Length(R[i]);
    end; //while 1
  end;

  exports
    Code name 'Code',
    BeforeBirthday name 'BeforeBirthday',
    ArToRom name 'ArToRom',
    RomToAr name 'RomToAr';

  begin
  end.
```

Лістинг . (html, txt)

Розберемо код. Насамперед, ми включили в розділ uses два додаткових модулі:

```
  uses
    Classes, DateUtils, SysUtils
    { you can add units after this };
```

Ці модулі були необхідні для роботи з датами - для одержання поточної дати, для розбирання дати на складові (рік, місяць, день) і на зворотну зборку, а також для визначення кількості днів від однієї дати до іншої.

Далі, у нас описані дві константи:

```
  const
    R: array[1..13] of string[2] =
    ('I', 'IV', 'V', 'IX', 'X', 'XL', 'L', 'XC', 'C', 'CD', 'D', 'CM', 'M');
    A: array[1..13] of Integer =
    (1, 4, 5, 9, 10, 40, 50, 90, 100, 400, 500, 900, 1000);
```

Ці константи являють собою масиви - перший рядковий, на два символи, другий - масив цілих чисел. Обидва масиви потрібні для заміни типу цифр. Наприклад, римська III відповідає арабської 3, римська X - це арабська 10, і так далі. Отут варто мати на увазі, що від'ємних римських цифр не буває, і що мінімальна цифра - одиниця. Крім того, тип integer має максимальне значення 2 147 483 647. Навряд чи вам буде потрібно переводити на римські більшу цифру - римські позначення не так популярні й використаються, в основному, у літературі, для позначення номера глави або розділу. Однак, пам'ятайте, що ми включаємо в бібліотеку тільки інструменти конвертування, а перевірку припустимих значень повинен робити програміст, що користується нашою DLL програмою.

Підемо далі. Першої в нас описана функція кодування / декодування рядків:

```
{цифрація/дешифрація}
function Code(s: PChar; Key: integer): PChar; stdcall;
var
  i: integer;
  ss: string;
begin
  ss:= s;
  for i:= 1 to length(s) do
    ss[i]:= char(Ord(ss[i]) xor Key);
  Result:= PChar(ss);
end;
```

Тут ми насамперед бачимо, що функція приймає параметри - рядок типу PChar і ціле число - ключ, і повертає також рядок PChar, хоча всередині самої функції використається тип String. Ще ми бачимо, що після оголошення імені функції вказується ключове слово stdcall, що означає, що для виклику цієї функції буде використана стандартна угода.

Щоб у програми можна було використати інструменти з динамічних бібліотек сторонніх розроблювачів, були розроблені спеціальні угоди по виклику процедур. Ці угоди визначають різні правила виклику підпрограм: як будуть передаватися параметри - через стек, через регістри, через динамічну пам'ять; хто відповідальний за очищення стека - зухвала або викликувана програма й т.д. Так, якщо

використається стек, то читання буде відбуватися праворуч ліворуч. Тобто, останні завантажені дані будуть лічені першими.

За замовчуванням, використається угода register. Його має сенс використати тоді, коли ви створюєте на Lazarus і DLL, і її програму, що використає. Такий виклик працює швидше, однак він майже не застосовується, тому що звичайно програмісти воліють використати більше гнучкі угоди stdcall і cdecl.

Виклики register і pascal передають параметри ліворуч праворуч, тобто перший параметр ліворуч обчислюється й передається в першу чергу, а останній параметр праворуч - обчислюється й передається останнім. Виклики cdecl і stdcall передають параметри навпаки, праворуч ліворуч. У таблиці нижче зазначені способи виклику процедур:

Таблиця 26.1. Відомості по угодах виклику підпрограм

Команда виклику	**Обробка параметрів**	**Відповідальний за очищення стека**	**Чи дозволена передача параметрів через регістри**
Register	Праворуч	Підпрограма	Так
Pascal	Праворуч	Підпрограма	Немає
Cdecl	Ліворуч	Зухвала програма	Немає
Stdcall	Ліворуч	Підпрограма	Немає

Зовсім влазити в нетрі машинної мови, мабуть, не вартує, запам’ятаєте тільки кілька рекомендацій. Для програм на Windows найчастіше використають угоду stdcall. А якщо ви створюєте DLL, що потім можуть використати Сі-програмісти, то вказуйте угоду cdecl.

Підемо далі. Команда

```
ss:= s;
```

не просто привласнює змінної ss передану в параметрі рядок, вона одночасно робить перетворення типів з PChar в String. Якщо ви не забули, такі перетворення можна робити явно, наприклад:

```
ss:= String(s);
s:= PChar(ss);
```

У нашій функції перетворення з PChar в String не зазначено, тому що воно відбувається за замовчуванням, однак зворотне перетворення все-таки було зробити необхідно:

```
Result:= PChar(ss);
```

Усередині підпрограми влаштований цикл

```
for i:= 1 to length(s) do
  ss[i]:= char(Ord(ss[i]) xor Key);
```

Тут, черговому символу рядка ss привласнюється результат роботи побітової операції XOR. До речі, шифрування тексту звичайно роблять саме за допомогою XOR, наш приклад тільки більше простій. Що в цьому випадку робить XOR? Оператор XOR використається для інвертування певних бітів числа (ще говорять, що XOR - операція що виключає АБО). Для окремих бітів оператор XOR працює в такий спосіб (дивитися ліворуч праворуч):

```
0 xor 0 = 0
0 xor 1 = 1
1 xor 0 = 1
1 xor 1 = 0
```

Тобто, якщо два аргументи рівні, XOR повертає False (для окремих бітів - нуль). А якщо аргументи розрізняються, XOR повертає True (або 1 для бітів).

Кожний символ таблиці ASCII має власне числове позначення. Так, англійська «A» має номер 65, англійська «B» - 66, і так далі, це все ми обговорювали в темі № 5. Комп'ютер працює тільки у двійковій системі, для нього «A» - це не десяткова 65, а двійкова 0100 0001. Отож, XOR обробляє не все число цілком, а окремо кожний його біт. У нашому прикладі ліворуч від XOR зазначений черговий символ рядка, а праворуч - число-ключ, від якого залежить зміна біта символу. З огляду на, що в таблиці ASCII може бути максимум, 255 символів, занадто великий ключ указувати не коштує, щоб уникнути помилок. 10 цілком достатньо, хоча можете й поекспериментувати (у мене після значення ключа 33 українські літери при шифрації починали спотворюватися, так що вибирайте значення від 1 до 30). Таким чином, ми кожний символ рядка кодуємо в зовсім інший символ. У більш складних системах шифрування ключ може змінюватися від біта до біта.

Коли всі символи рядка оброблені, рядок перетвориться в PChar, і результат повертається назад у програму.

Наступною в нас іде функція підрахунку кількості днів до наступного дня народження. День народження вказується в параметрі, що має тип TDateTime. Щоб підрахувати кількість днів до чергового дня народження, нам потрібно одержати нову дату - день і місяць дня народження, а рік ми повинні вказати поточний. Для цього ми спочатку «розбираємо» дату на складові:

```
DecodeDate(mydate, y, m, d);
```

Так ми одержали окремо, рік, місяць і день. Потім ми змінили рік на поточний:

```
y:= YearOf(Date);
```

Після чого зібрали дату назад. Але це тільки півсправи. Адже може бути три варіанти:

1. День народження сьогодні. У цьому випадку ми повертаємо 0 днів до дня народження.
2. Дня народження цього року ще не було. Ми повертаємо кількість днів від поточної дати до дати дня народження, для цього використаємо функцію DaysBetween, що повертає різницю в днях між двома зазначеними датами. До речі, при бажанні ви можете використати й інші подібні функції: YearsBetween - повертає кількість років між двома датами, MonthBetween - кількість місяців, WeeksBetween - кількість тижнів, HoursBetween - кількість годин, MinutesBetween - кількість хвилин, SecondsBetween - кількість секунд і MilliSecondsBetween - кількість мілісекунд. Тільки майте на увазі, що якщо вам потрібно одержати різницю в годинниках, хвилинах, секундах або миллисекундах, те замість Date, що повертає тільки поточну дату вам потрібно буде використати Now, що повертає поточну дату й час.
3. День народження цього року вже був. У цьому випадку нам потрібна нова дата - дата дня народження в наступному році. Для цього нам знову потрібно «розібрати» дату, додати до року одиницю й знову зібрати її. Після чого повертаємо кількість, що залишилася, днів від поточної дати до нової дати дня народження.

Функції перекладу римських цифр в арабські й назад, для економії місця пропоную розглянути самостійно - у нас адже лекція про

створення й виклик DLL, а не про синтаксис Паскаля. Якщо ви уважно вивчали попередні лекції, то зможете розібратися із синтаксисом. До цього ми не розглядали тільки процедуру декремента Dec, що зменшує значення аргументу. За замовчуванням, вона зменшує аргумент на одиницю. Або можна вказати аргумент і через кому, значення, на яке потрібно зменшити аргумент:

```
Dec(i);  //зменшує і на одиницю
Dec(N, A[i]);  //зменшує N на значення A[i]
```

З іншим синтаксисом ви повинні бути вже знайоми.

У самому кінці бібліотеки в нас зазначений список функцій, які можна буде викликати із зовнішніх програм:

```
exports
  Code name ‘Code’,
  BeforeBirthday name ‘BeforeBirthday’,
  ArToRom name ‘ArToRom’,
  RomToAr name ‘RomToAr’;
```

У бібліотеці може бути й більше процедур і функцій, у списку exports потрібно вказувати тільки ті, які будуть доступні ззовні. У цих же функцій указувалися й угоди викликів. У нашому випадку, всі наявні функції повинні бути доступні ззовні.

Тепер про спосіб описів списку. Директива

```
Code name ‘Code’,
```

говорить про те, що наша функція Code повинна викликатися із зовнішньої програми по імені Code. Для чого це зроблено? Допустимо, у зовнішній програми вже є функція Code, тоді ми одержимо конфлікт імен. У цьому випадку ми могли б викликати підпрограму під іншим ім’ям, наприклад:

```
Code name ‘NewCode’,
```

Тоді в зовнішній програми ми запитували б не функцію Code, а функцію NewCode.

Крім того, у списку exports можна вказувати передачу підпрограм не по іменах, а по індексах, наприклад, так:

```
exports
  Code, BeforeBirthday, ArToRom, RomToAr;
```

Тоді б підпрограмам автоматично були б привласнені індекси, від 1 до 4. Але індекси можна привласнити й примусово, за допомогою директиви index, наприклад, так:

```
exports
  Code index 1,
  BeforeBirthday index 2,
  ArToRom index 3,
  RomToAr index 4;
```

Що нам дають індекси? У середовищі Windows підпрограми по індексах викликаються ледве швидше, однак запам'ятовувати, під яким індексом перебуває потрібна підпрограма складніше. Крім того, використання індексації має сенс тільки в Windows, тому що інші ОС формують індекси автоматично. Загалом, рекомендується звертатися до підпрограм по іменах, як ми й зробили в нашій бібліотеці.

На цьому робота з динамічною бібліотекою закінчена. Оскільки кнопка **Run** на панелі інструментів для DLL недоступна, натисніть **<Ctrl+F9>** або виберіть команду «**Запуск -> Компілювати**». У результаті одержите готову бібліотеку - файл **MyFirstDLL.dll**.

Виклик DLL із зовнішньої програми

Бібліотеку DLL можна зв'язати з додатком двома способами:

- **статичним зв'язуванням**
- **динамічним зв'язуванням**

Статичне зв'язування найпростішому, воно має на увазі, що DLL буде завантажена відразу, як тільки додаток почне виконуватися. Це найпростіший спосіб використання DLL, звертання до процедур і функцій динамічної бібліотеки таке ж, як до звичайних процедур і функцій. Але є й мінуси використання статистичного зв'язування.

По-перше, при завантаженні програми повинні довантажуватися й всі використовувані DLL, а це збільшує час завантаження (при сучасній техніці, щоправда, ненабагато).

По-друге, щоб користувач міг працювати із програмою, він повинен мати всі ці використовувані DLL. Звичайно, так і є, але бувають і виключення. Можна було б роздати незареєстрованним користувачам усічену версію програми, без деяких ключових DLL. Тоді вони могли б користуватися програмою, але не всіма її можливостями, а тільки основними. Зареєстрованим користувачам можна було б додатково розсилати інші DLL. При статичному зв'язуванні це недоступно.

По-третє, при статичному зв’язуванні всі завантажені DLL займають пам’ять увесь час, що працює програма, поза залежністю від того, чи використає вона ці DLL у цей час, чи ні.

Динамічне зв’язування має на увазі, що необхідна DLL буде довантажуватися тільки тоді, коли потрібно виконати якусь процедуру або функцію з її. Після чого DLL можна вивантажити, звільнивши пам’ять. Динамічне зв’язування також має плюси й мінуси. Плюси в тім, що програму можна використати, не маючи всіх DLL. Завантаження програми буде здійснюватися швидше, оскільки DLL будуть довантажуватися в миру необхідності, а не разом із програмою.

Мінуси ж у тім, що через необхідність довантажувати й вивантажувати бібліотеку, буде збільшуватися час звертання до її функцій і процедур. Крім того, саме звертання до них буде набагато складніше.

Статичне зв’язування DLL

Розберемо спочатку більше простий спосіб. Відкрийте **Lazarus** з новим проєктом, ім’я форми зміните на **fMain**, в Caption напишіть «Статичне зв’язування DLL», Встановіть властивості BorderStyle в bsDialog, а Position в poDesktopCenter. Натисніть кнопку «**Зберегти всі**» і збережете проєкт під ім’ям **Proba** у папку **26-02**. Модуль форми назвіть як звичайно, Main.

Тепер скопіюйте файл **MyFirstDLL.dll** з папки **26-01** у папку **26-02**, щоб наш проєкт міг користуватися цієї DLL.

Далі, перейдіть у **Редактор коду**. Щоб ми могли використати функції й процедури DLL, їх потрібно оголосити так само, як в DLL. Оголошення потрібно робити відразу після ключового слова implementation, і після директиви процесору {$R *.lfm}:

```
implementation

{$R *.lfm}
function Code(s: PChar; Key: integer): PChar; stdcall;
        external ‘MyFirstDLL.dll’;
function BeforeBirthday(Birthday:TDateTime): Integer; stdcall;
        external ‘MyFirstDLL.dll’;
function ArToRom(N: Integer): PChar; stdcall;
        external ‘MyFirstDLL.dll’;
function RomToAr(s: PChar): Integer; stdcall;
        external ‘MyFirstDLL.dll’;
```

Як бачите, оголошення необхідних нам функцій таке ж, яким було в DLL, але також після оголошення була додана директива external:

external ‘MyFirstDLL.dll’;

яка означає, що ці функції програма повинна шукати у відповідному файлі динамічної бібліотеки, адже програма може використати трохи DLL. І це все, що потрібно для статичного зв’язування. Надалі ми зможемо звертатися до цих функцій, як д будь-яким інших.

Займемося компонентами. Встановіть на форму мітку TLabel, в Caption якої напишіть «Укажіть день народження:». Далі встановите компонент TDateEdit із вкладки **Misc Палітри компонентів**, тут користувач буде вводити або вибирати необхідну для перевірки дату народження. Властивість Name компонента перейменуйте в DE1. Правее встановите просту кнопку **TButton**, в Caption якої напишіть «До дня народження», а Name перейменуйте в bBeforeBirthday.

Нижче встановимо компоненти для перевірки перекладу формату цифр із арабських у римські, і навпаки. Спочатку встановимо митку, що пояснює, TLabel, в Caption якої напишемо «Укажіть ціле число від 1 до 2147483647:». Далі встановимо рядок TEdit, що перейменуємо в eNumbers, і очистимо властивість Text. Нижче встановимо дві прості кнопки **TButton**, які перейменуємо в bArToRom і bRomToAr, а в Caption цих кнопок напишемо відповідно, «Арабське в римське» і «Римське в арабське».

Наконец, займемося компонентами для шифрування тексту. Спочатку митку TLabel з текстом в Caption «Впишіть текст:». Далі рядок TEdit, що перейменуємо в eCode, і в якої очистимо властивість Text. Нижче встановимо просту кнопку **TButton**, перейменуємо її в bCode, і у властивості Caption напишемо «Шифрувати/дешифрувати». Зміните положення й розміри компонентів так, щоб ваша форма виглядала приблизно так:

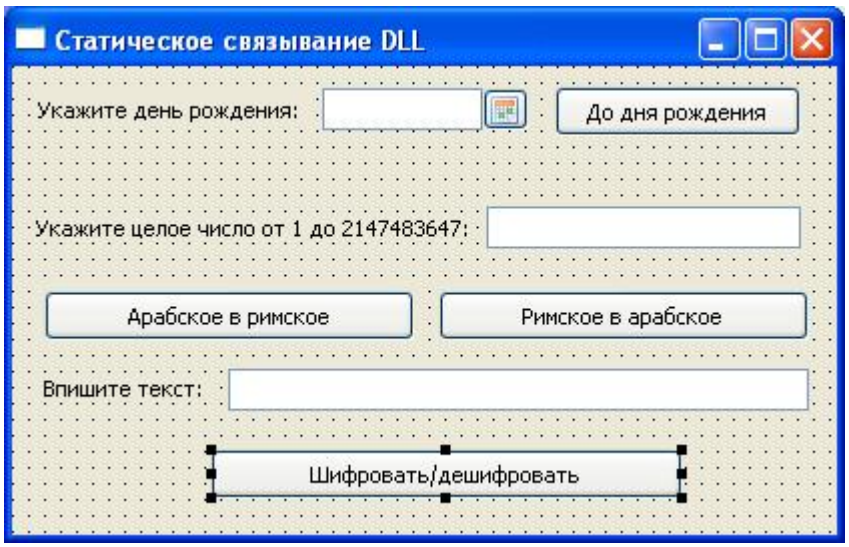


Рис. 26.2. Форма додатка

Нам залишилося написати код події OnClick для наших чотирьох кнопок, і випробувати в дії роботу з DLL. Код OnClick для кнопки **bBeforeBirthday** буде такою:

```
procedure TfMain.bBeforeBirthdayClick(Sender: TObject);
begin
  if DE1.Text = ‘‘ then exit;
  ShowMessage(‘До дня народження залишилося ‘ +
            IntToStr(BeforeBirthday(DE1.Date)) + ‘ днів’);
end;
```

Просто, не чи правда? Якщо користувач не вибрав дня народження, залишивши рядок DE1 порожній, ми виходимо із процедури, нічого не роблячи. Інакше ми виводимо повідомлення зі збірні із трьох частин рядком. Виклик функції з DLL:

```
IntToStr(BeforeBirthday(DE1.Date))
```

поверне нам у вигляді рядка кількість днів, що залишилися до дня народження.

Для кнопки **bArToRom** код події OnClick буде ще простіше:

```
procedure TfMain.bArToRomClick(Sender: TObject);
begin
  if eNumbers.Text = ‘‘ then exit;
  eNumbers.Text:= ArToRom(StrToInt(eNumbers.Text));
end;
```

Тут ми робимо єдину перевірку - чи не порожній рядок eNumbers. Майте на увазі, що в реальному додатку вам ще потрібна була б перевірка на вміст тексту: чи число це, чи ціле, чи входить воно в діапазон від 1 до 2 147 483 647? У нашій пробній програми для економії місця ми припускаємо, що користувач не буде вводити некоректних даних.

Для кнопки **bRomToAr** код схожий:

```
procedure TfMain.bRomToArClick(Sender: TObject);
begin
  if eNumbers.Text = ‘‘ then exit;
  eNumbers.Text:= IntToStr(RomToAr(PChar(eNumbers.Text)));
end;
```

У реальній програми вам також довелося б робити перевірку на коректність уведення даних римських цифр. Тут же ми будемо вважати, що користувач увів правильні римські цифри.

Ну й, нарешті, для кнопки **bCode** код буде таким:

```
procedure TfMain.bCodeClick(Sender: TObject);
begin
  if eCode.Text = ‘‘ then exit;
  eCode.Text:= Code(PChar(eCode.Text), 10);
end;
```

Вище я радив вам вибирати розмир ключа в діапазоні від 1 до 30, у виклику функції Code з DLL я вказав розмир ключа 10. От, властиво, і все. Збережете проєкт і запустіть його. Якщо ви всі зробили правильно, ваша програма буде працювати, як треба:

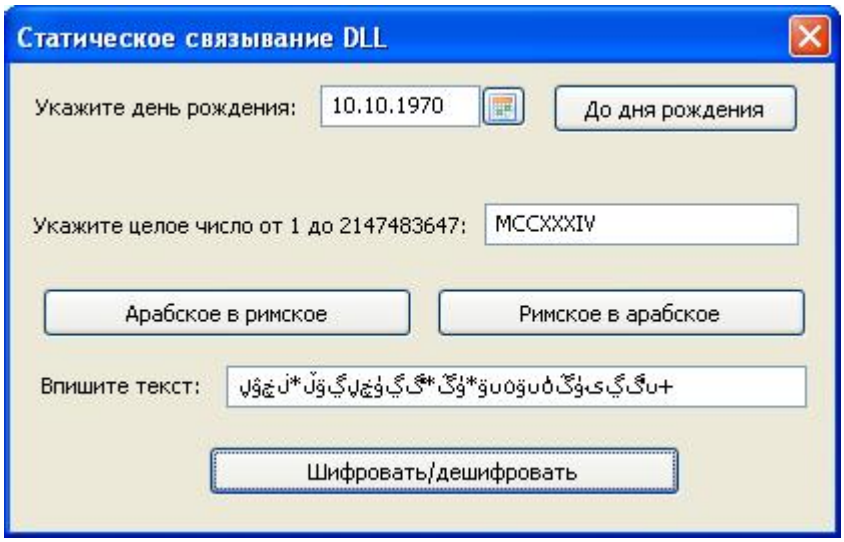


Рис. 26.3. Вікно робочої програми

3. Динамічне зв'язування DLL

Динамічне зв'язування відбувається набагато складніше статичного, причому динамічне зв'язування в Lazarus трохи відрізняється від такого ж зв'язування в Delphi, тому приклади Delphi отут працювати не будуть.

Отже, почнемо з того, що процедуру динамічного зв'язування можна розбити на чотири етапи:

1. Завантаження динамічної бібліотеки функцією LoadLibrary і одержання її дескриптора - числового ідентифікатора бібліотеки в системи. Такий ідентифікатор являє собою ціле число. Якщо завантаження бібліотеки було невдалої, то в дескриптор запишеться нуль.
2. Одержання адреси потрібної функції (або функцій) за допомогою GetProcAddress.
3. Безпосередньо робота з потрібною функцією з DLL.
4. Вивантаження DLL з пам'яті.

Причому ви можете виконувати всі дії по завантаженню бібліотеки на згадку і її вивантаженню як для виклику кожної процедури або функції окремо, так і для виклику групи підпрограм. Наприклад, користувач захотів виконати якусь операцію. Для цієї операції потрібно викликати, скажемо, десять підпрограм (процедур і функцій) з DLL. Буде нерозумно відкривати й закривати DLL окремо для кожної підпрограми, правильніше буде відкрити DLL, виконати всі ці десять підпрограм, а потім закрити DLL.

Але в нашому прикладі для наочності ми будемо завантажувати й вивантажувати DLL окремо для кожної операції.

Для цього додатка нам будуть потрібні точно такі ж компоненти на форми, як і в попередні. Якщо вам не лінь, то можна повернутися назад, і підготувати форму в такий же спосіб, як і в проєкту **Proba**. Однак можна зробити простіше. Завантажите в **Lazarus** проєкт **Proba** з папки **26-02**, для цього вам потрібно завантажити файл **Proba.lpi** або **Proba.lpr**. Далі виведіть на передній план **Редактор форми**. Клацніть правою кнопкою миші по вільному місцю форми й виберіть команду «**Виділити всі**». При цьому виявляться виділеними все компоненти форми. Потім виберіть команду головного меню «**Виправлення -> Копіювати**».

Тепер почнемо новий проєкт. Виберіть команду «**Файл -> Створити -> Додаток**», і у вас відкриється новий проєкт із порожньою формою. Розтягніть форму по ширині й висоті, щоб на ній легко вмістилися все компоненти, пізніше розміри форми можна буде підкорегувати. Тепер виберіть команду головного меню «**Виправлення -> Вставити**». При цьому всі потрібні компоненти з'являться на форми на тім же самим мисці. Крім того, вони збережуть свої імена та інші настроювання. Зкорегуйте розмір форми. Тепер перейменуйте форму в **fMain**, в Caption напишіть «Динамічне зв'язування DLL», в BorderStyle виберіть bsDialog, а в Position - poDesktopCenter. Збережете проєкт у папку **26-03** під ім'ям **Proba2**, модулю форми дайте ім'я Main. Не забудьте скопіювати файл **MyFirstDLL.dll** і в цю папку.

Тепер нам буде потрібно деяка підготовка перед тим, як ми почнемо динамічно зв'язувати нашу DLL. Насамперед, у розділ uses додайте ще один модуль - Dynlibs. Саме в ньому описані необхідні інструменти для динамічного підключення бібліотек.

Далі, перед глобальним розділом var нам потрібно описати типи наших функцій, що підключають, а в самому розділі var додати змін-дескриптор бібліотеки, а також по змінній на кожну функцію:

```
type
  TCode = function(s: PChar; Key: integer): PChar; stdcall;
  TBeforeBirthday = function(Birthday:TDateTime): Integer; stdcall;
  TArToRom = function(N: integer): PChar; stdcall;
  TRomToAr = function(s: PChar): Integer; stdcall;

var
  fMain: TfMain;
  My: THandle = 0; //для дескриптора бібліотеки
  Code: TCode;
  BeforeBirthday: TBeforeBirthday;
  ArToRom: TArToRom;
  RomToAr: TRomToAr;

implementation

{$R *.lfm}
```

Зверніть увагу, як ми описуємо тип функцій:

```
  TCode = function(s: PChar; Key: integer): PChar; stdcall;
```

Тип ми назвали TCode, ви можете дати інше ім’я, але для типів і класів традиційно прийнято починати ім’я з великої букви «Т». У цьому типі ми вказали, що створюється функція з такими то параметрами (як в DLL), що повертає тип PChar, і буде використати угоду stdcall. При цьому імені самої функції ми не вказуємо. У такий же спосіб ми створюємо ще три типи.

Далі, у глобальному розділі var ми додаємо таку змінну:

```
My: THandle = 0; //для дескриптора бібліотеки
```

Це - дескриптор. Коли ми завантажимо DLL на згадку, у цю змінну потрапить ідентифікатор нашої бібліотеки. Адже в пам’яті увесь час завантажена безліч самих різних DLL, використовуваних системою, і Windows якось потрібно розрізняти, до якої з них ви звертаєтеся. За замовчуванням відразу ж виставляємо нульове значення. Якщо спроба вважати DLL на згадку буде невдалої, наприклад, ця DLL відсутній, то дескриптор також буде нульовим.

Далі ми створюємо по змінній для кожної викликуваної з DLL функції. Наприклад, для функції Code ми створюємо змінну

```
Code: TCode;
```

Тип TCode ми описали вище, тут просто використаємо його ж. У принципі, ці змінні необов’язково робити глобальними, можна було описати їх у розділі var тієї процедури, звідки будемо викликати. Але в більших проєктах ту саму DLL-функцію звичайно викликають із безлічі різних процедур, тому зручніше описувати змінні, як глобальні.

Описавши дескриптор і змінні для функцій, ми зробили всю підготовчу роботу.

Переходимо до другого етапу - динамічному завантаженню DLL, використанні потрібної функції й вивантаженню DLL з пам’яті.

Згенеруйте подію OnClick для кнопки **bBeforeBirthday**. Код події відрізняється від минулого проєкту, він більше складний:

```
procedure TfMain.bBeforeBirthdayClick(Sender: TObject);
begin
  //виходимо, якщо порожньо:
  if DE1.Text = ‘‘ then exit;
  //відкриваємо бібліотеку й одержуємо її дескриптор:
  My:= LoadLibrary(‘MyFirstDLL.dll’);
  //виходимо, якщо помилка:
```

```
    if My = 0 then begin
      ShowMessage('Помилка відкриття бібліотеки MyFirstDLL.');
      exit;
    end;
    //одержуємо адресу потрібної функції:
    BeforeBirthday:=                TBeforeBirthday(GetProcAddress(My,
'BeforeBirthday'));
    //якщо функція прочиталася, обчислюємо, інакше виходимо:
    if @BeforeBirthday <> nil then  ShowMessage('До дня народження
залишилося ' +
       IntToStr(BeforeBirthday(DE1.Date)) + ' днів')
    else ShowMessage('Потрібна функція відсутня в бібліотеці');
    //вивантажуємо бібліотеку з пам'яті:
    FreeLibrary(My);
  end;
```

Давайте докладно розбирати весь цей код. Для початку, як і в минулому проєкті, ми зробили перевірку - є чи щось у рядку DE1, і якщо ні, те виходимо, тому що в подальших діях змісту немає. Якщо текст є (користувач вписав або вибрав дату народження), то виконується подальший код. Насамперед, ми завантажуємо на згадку бібліотеку, відразу ж одержуючи дескриптор на неї:

```
My:= LoadLibrary('MyFirstDLL.dll');
```

Функція LoadLibrary описана в модулі Dynlibs, що ми підключили в розділі uses, і намагається відкрити зазначену в параметрі бібліотеку. Якщо ім'я бібліотечного файлу зазначено без адреси, як у нашому випадку, то файл бібліотеки шукається в тій же папці, звідки запущена програма. Якщо його там немає, то бібліотека шукається в поточній папці (вона може відрізнятися від папки із програмою), потім у системних каталогах Windows і, нарешті, у папках, зазначених у системної змінної Path. Тому DLL краще встановлювати в папку із програмою, або в системну папку Windows, якщо цю DLL будуть використати кілька додатків. Такою папкою може бути, наприклад,

C:\Windows\system32

Якщо файл **MyFirstDLL.dll** був знайдений, то LoadLibrary завантажить його на згадку, а дескриптор передасть в My. Якщо по якійсь причині завантажити бібліотеку не вийшло, то LoadLibrary передасть в My значення 0. Тому далі ми робимо перевірку - чи

успішно завантажилася бібліотека? І якщо ні, те виводимо відповідне повідомлення, і виходимо:

```
if My = 0 then begin
  ShowMessage('Помилка відкриття бібліотеки MyFirstDLL.');
  exit;
end;
```

Далі, ми одержуємо адресу в пам'яті, де перебуває потрібна нам процедура:

```
BeforeBirthday:= TBeforeBirthday(GetProcAddress(My,
'BeforeBirthday'));
```

Для цього ми вказуємо ім'я типу цієї функції й використаємо функцію GetProcAddress. Ця функція працює так. Як параметри вона одержує дескриптор потрібної бібліотеки й назва функції (процедури), що нам потрібно відтіля викликати. Потім вона повертає адресу цієї функції в пам'яті, що попадає в нашу змінну BeforeBirthday. До речі, необов'язково давати цим зминним такі ж імена, як і у функцій, просто так удобней.

Якщо по якійсь причині адресу функції одержати не вдалося, наприклад, у пам'яті перебуває застаріла версія DLL, де ця функція відсутня, то GetProcAddress поверне значення nil, тобто, нічого. Саме тому ми робимо ще одну перевірку:

```
if @BeforeBirthday <> nil then  ShowMessage('До дня народження
залишилося ' +
    IntToStr(BeforeBirthday(DE1.Date)) + ' днів')
else ShowMessage('Потрібна функція відсутня в бібліотеці');
```

Оскільки змінна BeforeBirthday є, насправді, покажчиком, то в ній перебуває не функція з DLL з аналогічним ім'ям, а адреса цієї функції в пам'яті. Щоб подивитися, чи вказує на щось цей покажчик, ми використаємо оператор одержання адреси @, зазначений перед ім'ям покажчика:

```
if @BeforeBirthday <> nil
```

Якщо по даній адресі перебуває значення, не рівне nil, виходить, необхідна функція доступна, і ми виводимо повідомлення, як у попередньому проєкті:

ShowMessage(‘До дня народження залишилося ‘ + IntToStr(BeforeBirthday(DE1.Date)) + ‘ днів’)

Інакше функція недоступна, про що ми й повідомляємо користувачеві, не звертаючись при цьому до функції:

else ShowMessage(‘Потрібна функція відсутня в бібліотеці’);

У принципі, у нашому невеликому проєкті можна було б обійтися й без всіх цих численних перевірок, але я хотів показати, яким образом динамично підключаються DLL у реальних проєктах. Це на перший погляд весь цей код здається складним, а коли розберетеся й випробуйте його, він стане досить зрозумилим. Під кінець ми вивантажуємо DLL з пам’яті:

FreeLibrary(My);

Після цього, звертатися до її функцій уже буде не можна. Код звертання до інших функцій ми так докладно розбирати не будемо, оскільки він аналогічний розглянутому вище коду. Код події OnClick кнопки **bArToRom** наступний:

```
procedure TfMain.bArToRomClick(Sender: TObject);
begin
  if eNumbers.Text = ‘‘ then exit;
  //відкриваємо бібліотеку й одержуємо її дескриптор:
  My:= LoadLibrary(‘MyFirstDLL.dll’);
  //виходимо, якщо помилка:
  if My = 0 then begin
    ShowMessage(‘Помилка відкриття бібліотеки MyFirstDLL.’);
    exit;
  end;
  //одержуємо адресу потрібної функції:
  ArToRom:= TArToRom(GetProcAddress(My, ‘ArToRom’));
  //якщо функція прочиталася, обчислюємо, інакше виходимо:
  if @ArToRom <> nil then eNumbers.Text:= ArToRom(StrToInt(eNumbers.Text))
  else ShowMessage(‘Потрібна функція відсутня в бібліотеці’);
  //вивантажуємо бібліотеку з пам’яті:
  FreeLibrary(My);
end;
```

Код події OnClick кнопки **bRomToAr** такий:

```
procedure TfMain.bRomToArClick(Sender: TObject);
begin
  if eNumbers.Text = ‘‘ then exit;
  //відкриваємо бібліотеку й одержуємо її дескриптор:
  My:= LoadLibrary(‘MyFirstDLL.dll’);
  //виходимо, якщо помилка:
  if My = 0 then begin
    ShowMessage(‘Помилка відкриття бібліотеки MyFirstDLL.’);
    exit;
  end;
  //одержуємо адресу потрібної функції:
  RomToAr:= TRomToAr(GetProcAddress(My, ‘RomToAr’));
  //якщо функція прочиталася, обчислюємо, інакше виходимо:
  if @RomToAr <> nil then
     eNumbers.Text:= IntToStr(RomToAr(PChar(eNumbers.Text)))
  else ShowMessage(‘Потрібна функція відсутня в бібліотеці’);
  //вивантажуємо бібліотеку з пам’яті:
  FreeLibrary(My);
end;
```

І, нарешті, код OnClick кнопки **bCode** такий:

```
procedure TfMain.bCodeClick(Sender: TObject);
begin
  if eCode.Text = ‘‘ then exit;
  //відкриваємо бібліотеку й одержуємо її дескриптор:
  My:= LoadLibrary(‘MyFirstDLL.dll’);
  //виходимо, якщо помилка:
  if My = 0 then begin
    ShowMessage(‘Помилка відкриття бібліотеки MyFirstDLL.’);
    exit;
  end;
  //одержуємо адресу потрібної функції:
  Code:= TCode(GetProcAddress(My, ‘Code’));
  //якщо функція прочиталася, обчислюємо, інакше виходимо:
  if @Code <> nil then eCode.Text:= Code(PChar(eCode.Text), 10)
  else ShowMessage(‘Потрібна функція відсутня в бібліотеці’);
  //вивантажуємо бібліотеку з пам’яті:
  FreeLibrary(My);
```

```
end;
```

Збережіть проєкт, запустіть його на виконання й переконаєтеся, що робота динамично зв’язаної DLL нічим не відрізняється від статично зв’язаної DLL. Уповільнення викликів функцій якщо і є, то практично не помитні. Який із цих методів вибирати, залежить від конкретної ситуації й переваг програміста. Я, примиром, звичайно використаю статичне зв’язування з DLL.

Тема 27. Тестування й налагодження

1. Тестування й налагодження програми
2. Синтаксичні помилки
3. Помилки часу виконання

1. Тестування й налагодження програми

Чим більше досвіду має програміст, тим менше помилок у коді він робить. Але, хочете вірте, хочете ні, навіть самий досвідчений програміст все-таки допускає помилки. І будь-яке сучасне середовище розробки програм повинна мати власні інструменти для налагодження додатків, а також для своєчасного виявлення й виправлення можливих помилок. Програмні помилки на програмістському сленгу називають багами (англ. bug - жук), а програми налагодження коду - дебаггерами (англ. debugger - наладчик). Lazarus, як сучасне середовище розробки додатків, має власний вбудований наладчик, роботу з яким ми розберемо на цій темі.

Помилки, які може допустити програміст, умовно діляться на три групи:

1. Синтаксичні
2. Часу виконання (run-time errors)
3. Алгоритмічні

2. Синтаксичні помилки

Синтаксичні помилки легше всього виявити й виправити - їх виявляє компілятор, не даючи скомпілювати й запустити програму. Причому компілятор установлює курсор на помилку, або після її, а у вікні повідомлень виводить відповідне повідомлення, наприклад, таке:

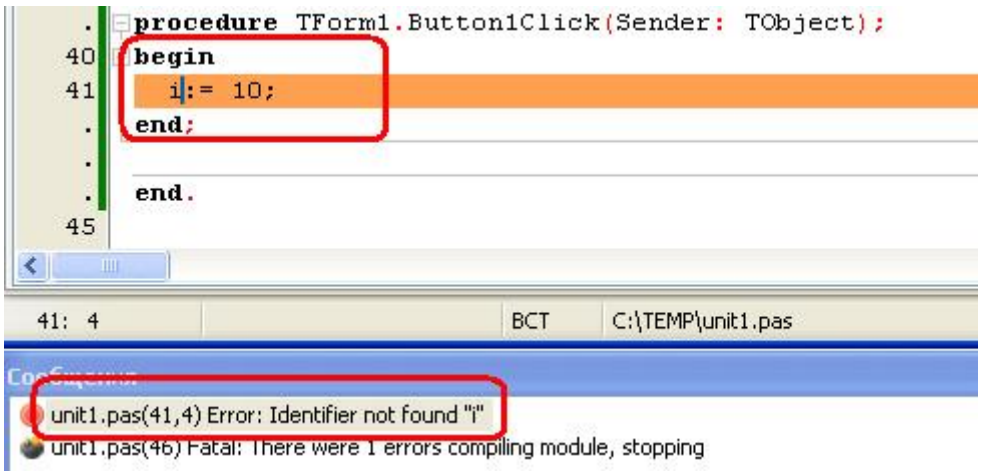


Рис. 27.1. Знайдена компілятором синтаксична помилка - немає оголошення змінної *i*

Подібні помилки можуть виникнути при неправильному написанні директиви або імені функції (процедури); при спробі звернутися до змінної або константи, що не оголошена (рис. 27.1); при спробі викликати функцію (процедуру, змінну, константу) з модуля, що не був підключений у розділі uses; при інших аналогічних недоглядах програміста.

Як уже говорилося, компілятор при знаходженні подібної помилки припиняє процес компіляції, виводить повідомлення про знайдену помилку й встановлює курсор на допущену помилку, або після її. Програмістові залишається тільки внести виправлення в код програми й виконати повторну компіляцію.

3. Помилки часу виконання

Помилки часу виконання (run-time errors) теж, як правило, легко переборні. Вони звичайно проявляються вже при перших запусках програми, або під час тестування. Якщо таку програму запустити із середовища Lazarus, то вона скомпилируется, але при спробі завантаження, або в момент здійснення помилки, призупинить свою роботу, вивівши на екран відповідне повідомлення. Наприклад, таке:



Рис. 27.2. Повідомлення Lazarus про помилку часу виконання

У цьому випадку програма при завантаженні повинна була вважати на згадку відсутній текстовий файл **MyFile.txt**. Оскільки програма викликала помилку, вона не запустилася, але в середовищі Lazarus процес налагодження триває, про що свідчить повідомлення в дужках у заголовку головного меню, після назви проєкту. Програмістові в подібних випадках потрібно скинути наладчик командою меню «**Запуск -> Скинути отладчик**», після чого можна продовжити роботу над проєктом.

Помилка часу виконання може виникнути не тільки при завантаженні програми, але й під час її роботи. Наприклад, якби спроба читання неіснуючого файлу була зроблена не при завантаженні програми, а при натисканні на кнопку, то програма б нормально запустилася й працювала, поки користувач не натисне на цю кнопку.

Якщо програму запустити із самої Windows, при виникненні цієї помилки з'явиться таке ж повідомлення. При цьому якщо нажати «**OK**», програма навіть може запуститися, але коректно працювати однаково не буде.

Швидкості часу виконання бувають не тільки явними, але й неявними, при яких програма продовжує свою роботу, не виводячи ніяких повідомлень, а програміст навіть не догадується про наявність помилки. Прикладом неявної помилки може служити так званий **витік пам'яті**. Витік пам'яті виникає у випадках, коли програміст забуває звільнити виділену під об'єкт пам'ять. Наприклад, ми повідомляємо змінну типу TStringList, і працюємо з нею:

```
begin
  MySL:= TStringList.Create;
  MySL.Add('Новий рядок');
end;
```

У даному прикладі програміст допустив типову для початківців помилку - не звільнив клас TStringList. Це не приведе до збою або аварійного завершення програми, але в підсумку можна даремно витратити дуже багато пам'яті. Звичайно, ця пам'ять буде звільнена після вивантаження програми (за цим стежить операційна система), але витік пам'яті під час виконання програми теж може привести до неприємних наслідків, споживаючи усе більше й більше ресурсів і

зайво навантажуючи процесор. У подібних випадках після роботи з об’єктом програмістові потрібно не забувати звільняти пам’ять:

```
begin
  MySL:= TStringList.Create;
  MySL.Add(‘Новий рядок’);
  ...; //робота з об’єктом
  MySL.Free; //звільнили об’єкт
end;
```

Однак помилки часу виконання можуть трапитися й під час роботи з об’єктом. Якщо є такий ризик, програміст повинен не забувати про можливість обробки виняткових ситуацій. У цьому випадку вищенаведений код правильней буде оформити в такий спосіб:

```
begin
  try
    MySL:= TStringList.Create;
    MySL.Add(‘Новий рядок’);
    ...; //робота з об’єктом
  finally
    MySL.Free; //звільнили об’єкт, навіть якщо була помилка
  end;
end;
```

Отже, щоб уникнути помилок часу виконання програміст повинен не забувати робити перевірку на правильність уведення користувачем припустимих значень, містити небезпечний код у блоки try...finally...end або try...except...end, робити перевірку на існування файлу, що відкриває, функцією FileExists і взагалі дотримувати передбачливості у всіх слабких місцях програми. Не покладайтеся на користувача, адже недарма говорять, що якщо в програми можна припуститися помилки, користувач цю можливість неодминно знайде.

Алгоритмичні помилки

Якщо ви не допустили ні синтаксичних помилок, ні помилок часу виконання, програма скомпилировалась, запустилася й працює нормально, то це ще не означає, що в програми немає помилок. Переконатися в цьому можна тільки в процесі її тестування.

Тестування - процес перевірки працездатності програми шляхом уведення в неї різних, навіть навмисно помилкових даних, і наступній контрольній перевірці виведеного результату.

Якщо програма працює правильно з одними наборами вихідних даних, і неправильно з іншими, то це свідчить про наявність алгоритмичної помилки. Алгоритмичні помилки іноді називають логічними, звичайно вони пов’язані з невірною реалізацією алгоритму програми: замість «+» помилково поставили «-», замість «/» - «*», замість розподілу значення на 0,01 розділили на 0,001 і т.п. Такі помилки звичайно не виявляються під час компіляції, програма нормально запускається, працює, а при аналізі виведеного результату з’ясовується, що він невірний. При цьому компілятор не вкаже програмістові на помилку - щоб знайти й усунути її, доводиться аналізувати код, пошагово «прокручувати» його виконання, стежачи за результатом. Такий процес називається налаштуванням.

Налагодження - процес пошуку й усунення помилок, частіше алгоритмичних. Хоча yfладчик дозволяє впоратися й з помилками часу виконання, які не виявляються явно.

Робота з отладчиком

Давайте від теорії перейдемо до практики. Завантажите **Lazarus** з новим проєктом, встановіть на форму просту кнопку й збережіть проєкт у папку **27-01**. Імена проєкту, форми, модуля й кнопки змінювати не потрібно, залишіть імена, дані за замовчуванням.

Далі, згенеруйте подію OnClick для кнопки, у якому напишіть наступний код:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
  st: TStringList;
begin
  //створюємо список рядків:
  st:= TStringList.Create;
  try
    //генеруємо список:
    for i:= -3 to 3 do begin
      st.Append('10/'+IntToStr(i)+'='+FloatToStr(10/i));
    end;
    //виводимо список на екран:
    ShowMessage(st.Text);
  finally
    //st буде звільнена навіть у випадку run-time помилки:
    st.Free;
```

```
  end;
end;
```

Що ми отут робимо? Цілочисельну змінну і використаємо як лічильник для циклу for. Цикл робимо від -3 до 3, тобто, 7 разів, включаючи нуль. У тілі циклу ми ділимо 10 на значення і, результат оформляємо у вигляді рядка й додаємо до списку рядків st. Вище говорилося, що подібні дії потрібно містити в блок обробки виняткових ситуацій try-finally-end, що ми й зробили.

Якщо ви уважно вивчали курс, то неозброєним оком бачите, що при четвертому проході циклу відбудеться помилка часу виконання - розподіл 10 на нуль. Такий очевидний приклад найбільше підходить для знайомства з убудованим отладчиком, тому що ви вже знаєте, де буде помилка, і зможете проаналізувати роботу отладчика. Тому прикинемося, що не підозрюємо про помилку.

Отже, програму ми написали, зберегли, пора її компілювати. Натисніть кнопку «**Запустити**» на **Панелі керування** (або **<F9>**). Програма нормально скомпилировалась і запустилася. Натиснемо кнопку **Button1**. І відразу одержуємо помилку:

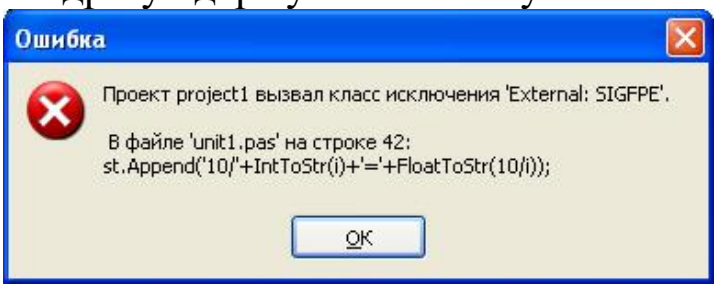


Рис. 27.3. Повідомлення Lazarus про помилку

Судячи з повідомлення, помилка відбулася під час виконання коду 42-й рядка. Добре, натиснемо «**ОК**» і командою «**Запуск -> Скинути отладчик**» припинимо виконання програми. Повернемося до коду й проаналізуємо 42-й рядок (якщо ви додавали порожні рядки, то у вас буде інший номер):

```
st.Append(‘10/’+IntToStr(i)+’=‘+FloatToStr(10/i));
```

Ну що, нічого криминального отут ні, чому ж відбулася помилка? Код вірний і повинен був нормально виконуватися…Коли ви заходите в подібний глухий кут, допомогти вам може здоровий глузд і убудований отладчик. Здоровий глузд говорить, що помилка відбулася десь у тілі циклу for. А щоб скористатися отладчиком, потрібно призупинити виконання програми на цьому циклі, щоб потім

построчно його продовжити. Для зупинки роботи програми служать так називані **крапки зупинки** (англ. breakpoints).

Крапки останова - це рядка, перед виконанням яких отладчик припиняє виконання програми, і чекає ваших подальших дій.

Ви можете встановити одну таку крапку або трохи, у різних частинах коду. Оскільки помилка виникає в 42-й рядку, краще буде призупинити виконання на попередньої, 41-й рядку. Переведіть курсор на цей рядок, на будь-яке її місце.

Установити крапку останова можна різними способами:

- Командою головного меню «**Запуск -> Додати крапку останова -> Крапка останова у вихідному коді**». У вікні, що відкрилися, «**Параметри крапки** останова» нажати «ОК».
- Клікнути по рядку правою кнопкою, і в спливаючому меню вибрати «**Налагодження -> Перемкнути крапку останова**».
- Нажати «гарячу клавішу» **<F5>**.
- Клацнути по потрібному рядку в лівій частині **Редактора коду**, де зазначені номери рядків.

Останні два способи найбільш зручні, але вибирати вам. У кожному разі, рядок із установленою крапкою останова офарбиться червоними кольорами:



Рис. 27.4. Рядок із крапкою останова

Зняти крапку останови зручніше також останніми двома способами. Крапка останови в нас є, знову натискаємо кнопку «**Запустити**». Програма починає свою роботу, натискаємо кнопку «**Button1**».

Тепер програма не вивела помилку, а призупинила свою роботу й вивела на передній план **Редактор кодів** з виділеної сірими кольорами рядком, що у цей момент готується до виконання:

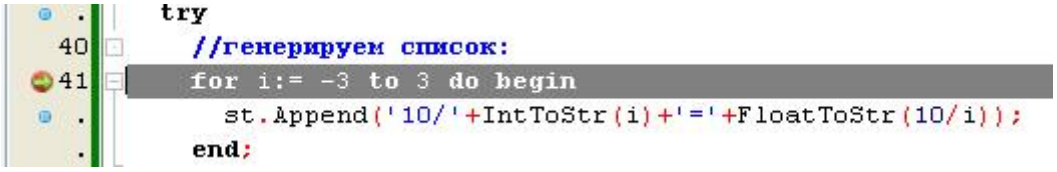


Рис. 27.5. Рядок, що буде виконий далі

Отут дуже важливо розуміти, що програма була зупинена ДО виконання цього рядка, а не ПІСЛЯ її. Тобто, у даний момент змінної і ще не привласнене значення -3. Далі, ми можемо виконувати з отладчиком різні дії, які зібрані в розділі головного меню «**Запуск**». Звичайно потрібне покрокове виконання програми. Для цього можна використати команду «**Запуск -> Крок в обхід**» (або **<F8>**), «**Запуск -> Крок із входом**» (або **<F7>**) або «**Запуск -> Крок з виходом**» (або **<Shift+F8>**). «**Крок в обхід**» означає, що якщо в коді буде зустрінутий виклик якої-небудь функції або процедури, отладчик виконає їх і зупиниться на наступному послу виклику рядку. При виборі «**Крок із входом**», отладчик також пошагово буде виконувати й викликувані функції-процедури. «**Крок з виходом**» має на увазі, що якщо в рядку немає викликів функцій, то зупинки відбуваються, як при «**Крок в обхід**». Якщо в рядку є вираження, то зупинка відбувається спочатку перед рядком, потім перед обчисленням кожної функції, щоб ми мали можливість переглянути значення параметрів, переданих у функцію.

У нас викликів функцій ні, тому ми можемо скористатися як **<F7>**, так і **<F8>** (найчастіше використають **<F8>** - Крок в обхід).

Отже, натиснемо **<F8>**, і наладчик виконає рядок із крапкою останови, і виділить сірим наступний рядок. Снову натиснемо **<F8>**, і знову буде виділений цей рядок - був виконаний крок циклу. Нажавши кілька разів **<F8>**, ми доможемося появи на екрані все тієї ж помилки, що заблокує подальше виконання програми. Стає зрозуміло, що цикл нормально виконується кілька проходів, після чого все-таки виникає помилка. Включаємо логікові: всередині циклу в нас змінюється тільки змінна **i**, виходить, якась помилка пов’язана з нею. А як довідатися, як саме?

Тут нам допомога! Ще один корисний інструмент наладчика - спостереження за значеннями змінних. Скиньте програму командою «**Запуск -> Скинути отладчик**». Тепер знову натисніть кнопку «**Запустити**», а потім знову кнопку «**Button1**». Наладчик знову призупинив виконання програми на рядку із циклом, однак не поспішаєте натискати **<F8>**. Для початку, додамо спостереження над змінної **i**. Робиться це командою «**Запуск -> Додати спостереження**», що була недоступна, поки програма не почала виконуватися. У рядку «**Вираження**» вкажіть змінну **i**, і натисніть «**ОК**»:

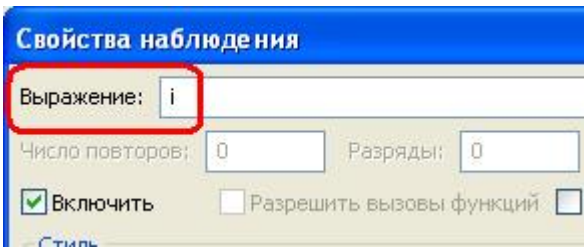


Рис. 27.6. Встановлення спостереження за змінною

Тепер наладчик спостерігає за значеннями змінної i, але нам від цього не легше – ми ж не бачимо цих значень! Щоб їх побачити, потрібно вивести на екран вікно **Списку спостережень**. Робиться це командою «**Вид -> Вікна налагодження -> Вікно спостережень**» або «гарячими клавішами» **<Ctrl+Alt+W>**.

Розташуєте вікно нижче **Редактор коду**, на місце **Вікна повідомлень**. У цьому вікні ви зможете бачити вираження - нашу змінну i, і її поточне значення. Щоб показати роботу з наладчиком більш наочно, давайте додамо ще одне вираження, за яким будемо спостерігати. Виберіть «**Запуск -> Додати спостереження**» і в рядку «**Вираження**» укажіть не просто змінну, а вираження

10/i

яке в нас має обчислюватися всередині циклу. У вікні **Списку спостереження** ви побачите й змінну i, і вираження, а також їхнього значення:

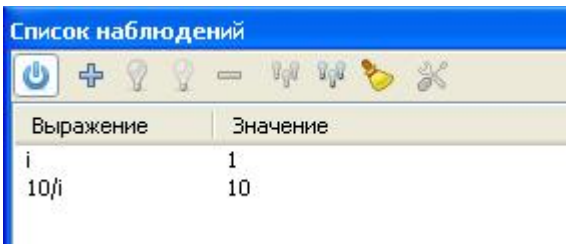


Рис. 27.7. Вікно Списку спостережень

Оскільки змінної i ще не було їй привласнено значення -3, то в колонку значень ви, швидше за все, побачите 1, яким за замовчуванням була проініціалізована наша змінна. Відповідне значення буде й у виразу. Тепер ми готові рухатися далі. Натискаємо **<F8>**. У **Списку спостережень** відразу ж змінилася картина - і тепер дорівнює -3, а вираз -3,3333…

Натискаємо **<F8>** ще раз. Знову значення змінилися, тепер i = -2, а вираження = -5. Ми розуміємо, що цикл працює, і два його кроки були

зроблені. Натискаємо **<F8>** ще два рази. Зараз змінна містить нуль, а значення вираження вказує «inf». Однак рядок з обчисленням ще не був виконаний, не забуваємо про це. Снову натискаємо **<F8>**, і знову одержуємо помилку. А в значеннях змінної й вираження бачимо слово «evaluating», що переводиться, як «оцінка». Тепер ми наочно бачимо, що в рядку

```
st.Append('10/'+IntToStr(i)+'='+FloatToStr(10/i));
```

виникає помилка, коли змінна і дорівнює нулю. І отут уже нескладно здогадатися, чому ця помилка виникає - тому що відбувається спроба розподілу 10 на 0.

Це можна перевірити, пропустивши виконання обчислення, коли i=0. Закрийте вікно з помилкою, скиньте наладчик. Знову натисніть кнопку «**Запустити**», і кнопку «**Button1**». Снову виведіть вікно **Списку спостереження**. Натискайте **<F8>**, поки і не стане 0, а вираз - inf. Тепер, у **Вікні спостережень** клацніть правою кнопкою миші по рядку зі змінною і, і в спливаючому меню виберіть команду «**Обчислити/Змінити**». У вікні, що відкрилося, ви побачите рядок «Вираз», де буде зазначена змінна і. У полі «**Результат**» буде зазначено значення 0. А в рядку «**Нове значення**» нам потрібно вказати значення, що ми бажаємо примусово привласнити змінної. Отут укажемо 1 і натиснемо **<Enter>** або кнопку «**Змінити**». У поле «**Результат**» значення повинне змінитися на 1. Снову натиснемо **<F8>**, значення і зміниться на 2, помилки не буде. Нажавши **<F8>** ще кілька разів, ми доберемося до кінця програми й побачимо повідомлення, що вона й повинно було вивести за нашим задумом:

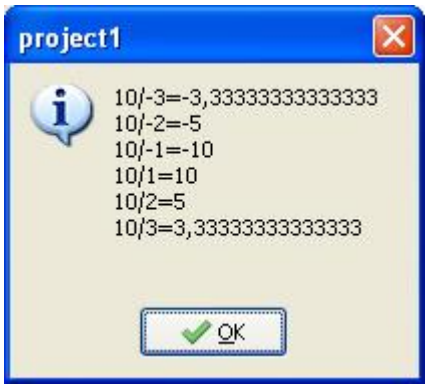


Рис. 27.8. Результуюче повідомлення програми

Як бачите, обчислення, де *і* дорівнює нулю, було пропущено.

Вбудований наладчик має й інші інструменти, з якими ви самі зможете згодом освоїтися, експериментуючи з ними.

Тема 28. Створення довідкової системи

1. Створення довідкової системи
2. Створення файлу проєкту
3. Створення повнотекстового пошуку

1. Створення довідкової системи

Будь-який серйозний проєкт повинен бути забезпечений докладною довідковою системою, що дозволяє користувачеві розібратися з тим, як працює ваша програма. Адже якби не «хелпи», ми з вами навряд чи змогли самостійно освоїти яку-небудь складну програму. І в цій лекції ми навчимося створювати довідкові системи й підключати їх до проєкту. Нагадаю, що хоч Lazarus і є крос-платформним середовищем розробки програм, ми освоюємо роботу з ним під керуванням ОС Windows. Якщо ви користуєтеся Lazarus у середовищі Linux, Unix або інший не Windows ОС, то рекомендації із цієї лекції вам навряд чи підійдуть. У цьому випадку, вам доведеться пошукати даний матеріал самостійно.

Отже, довідка може бути двох форматів: *.hlp і *.chm. Обидва формати розроблені корпорацією Microsoft. Формат HLP є застарілої, хоча й зустрічається дотепер, але в даний момент найпоширенішим форматом довідки в середовищі Windows є CHM, його й розглянемо.

CHM (Microsoft Compressed HTML Help, Microsoft Compiled HTML Help, CHM) - формат файлів контекстної довідки, розробленої Microsoft в 1997 році як замина формату WinHelp - HLP. Файл CHM є стислим набором HTML-сторінок, свого роду, електронною книгою, і може містити весь набір Web-сторінок: текст, таблиці, малюнки, посилання, медиа-файлы, та інше, що робить таку довідкову систему потужним інструментом. Всі вхідні в CHM-довідку файли стислі алгоритмом LZX, що робить довідку більше компактної.

В Інтернеті є згадка, що до складу Lazarus входить інструмент для створення CHM-файлів. І справді, цей інструмент є за адресою (папки установки Lazarus у мене створювалися за замовчуванням):

C:\lazarus\tools\chmmaker

За цією адресою є проєкт, що по ідеї, потрібно скомпілювати, і одержати готовий інструмент. Втім, досить глянути на вікно проєкту,

щоб зрозумити, наскільки даний інструмент примітивний. Говорити серйозно про створення з його допомогою довідкової системи не доводиться. А виходить, нам потрібно знайти інший інструмент.

Цей інструмент повинен бути:

- безкоштовним;
- досить потужним для створення довідок будь-якої складності.

І отут на думку відразу приходять дві програми: HTM2CHM і Microsoft HTML Help Workshop. Перша написана російським програмістом Ярославом Кириловим, має російським і досить дружелюбний інтерфейс. Функціонально вона програє Microsoft HTML Help Workshop. Крім того, файли, що компілюються програмою CHM- виходять трохи більшого розміру. Однак якщо потрібно проста невелика довідка, то кращого інструмента вам не знайти. Ми не будемо розглядати програму HTM2CHM, оскільки вона досить проста, вам неважко буде освоїти її самостійно. Розглянемо більше складний інструмент - Microsoft HTML Help Workshop. Як видно з назви, програма розроблена корпорацією Microsoft, є безкоштовним, вільно розповсюджуваним продуктом. Скачати її можна із сайту виробника, я знайшов неї за адресою: http://www.microsoft.com/en-us/download/details.aspx?id=21138

Це HTML Help Workshop версії 4.74.8702 - програма для створення довідкової системи.

HTML Help Workshop (hhw.exe) - програма, що володіє досить більшими можливостями, хоча й непроста в експлуатації, і не має російського інтерфейсу. Програма дозволяє створювати всі необхідні файли довідки, з яких потім компілюється єдиний проєкт.

Давайте взагалі, розберемося, що являє собою CHM-довідка. По суті, це електронна інтерактивна книга, що дозволяє використати зміст, індекси (тэги), повнотекстовий контекстний пошук, обрані статті довідки. І все це створюється на основі окремих HTML-файлів. І от тепер ми підходимо до іншому питанню: що таке HTML? **HTML** - Hyper Text Markup Language - Мова розмитки гіпертексту, основа, на якій будуються всі Web-сайти, ті сами веб-странички, які ми завантажуємо у свої браузери. Виходить, щоб створювати довідкову систему, потрібно вивчити ще який те HTML? Загалом, так, якщо ви будете використати тільки вбудовані засоби HTML Help Workshop. Веб-сторінку можна написати в будь-якому текстовому редакторі, хоч у Блокноті, якщо ви знаєте розмітку HTML. Однак звичайно, є й більше прості шляхи.

Існує безліч візуальних редакторів, які дозволяють зберігати вміст у вигляді HTML-файлів. У першу чергу приходить на розум офісна програма MS Word. Так, вона також дозволяє створювати HTML-файли, однак не раджу її використовувати: усередині таких файлів дуже багато зайвих тегів - сміття. Крім того, посилання на зображення в HTML-файлах повинні бути представлені тегом img, наприклад:

<img src=«logo.png»>

а MS Word вставляє зображення в колекцію Shapes.

Іншим варіантом в Інтернеті нерідко радять програму з того ж офісу **MS FrontPage**. Що ж, у перших версіях ця програма дійсно, була гарна: досить проста, дозволяла створювати веб-сторінки візуально, не жадала від користувача знань HTML. Однак згодом, FrontPage (як і інша продукція Microsoft) усе більше ускладнювалася, у неї додавалися всі нові, невідомо кому потрібні функції, і тепер MS FrontPage - неповороткий монстр, для вивчення якого буде потрібно досить важкий підручник.

Є й третій, більше вдалий варіант. На просторах Інтернету існує багато візуальних HTML-редакторів, що не вимагають знань HTML. Є й платні програми, і безкоштовні. Є досить складні системи, начебто Macromedia Dreamweaver, що дозволяють будувати цілі сайти, є програми попроще.

Для створення HTML-файлів я досить рекомендую простий і безкоштовний візуальний HTML-редактор з російським інтерфейсом NVU. Цей редактор ви можете скачати із сайту розроблювача: http://nvu.mozilla-russia.org/

або з мого сайту: http://lazarusprog.tk/files/nvu.rar

Це NVU 1.0 - Візуальний HTML редактор (**WYSIWYG-редактор**, від англ. What You See Is What You Get, «що бачиш, то й одержиш»). Цей редактор не вимагає знань HTML, досить простий в освоєнні й має докладну довідку російською мовою:

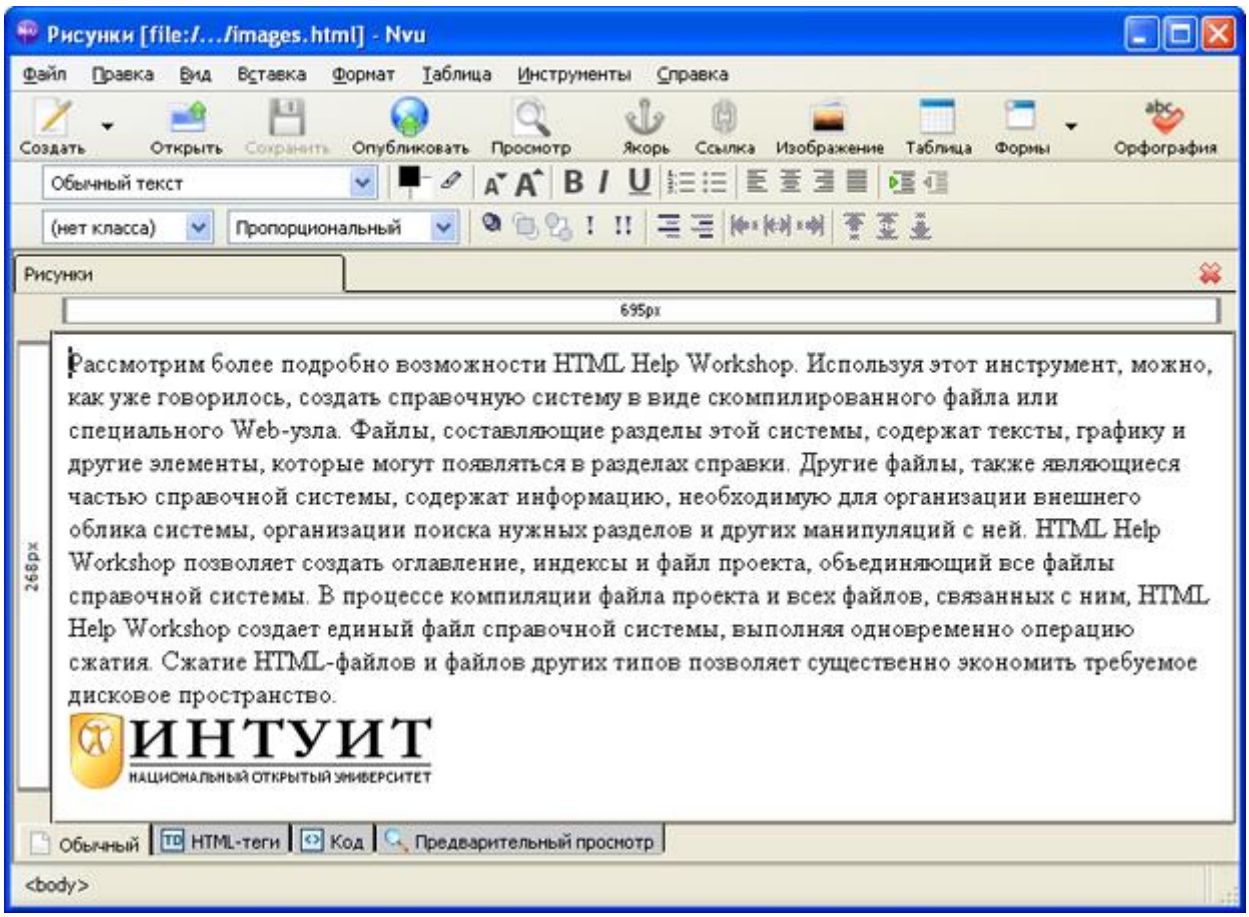


Рис. 28.1. HTML-редактор Nvu

Розглядати створення веб-сторінок у цьому редакторі ми не будемо, там все достаточно просто й прозоро, а лекція й без того вийде великий. Зазначу тільки, що імена HTML-файлам краще давати латинськими символами й не використати пробіли, тобто, замість «Команди меню.html» файл буде краще назвати «CommandsMenu.html». Справа в тому, що якщо ви будете розміщувати на якихось сторінках керівництва посилання на інші сторінки довідника, то HTML Help Workshop може дати невірні результати з іменами файлів українською мовою. Пробіли в імені файлу теж можуть послужити погану службу.

Тепер повернемося до нашої довідки. Сподіваюся, ви зберегли проєкт Блокнота-шифрувальника з 18-й лекції? Для цього проєкту ми й будемо створювати довідкову систему. Насамперед, нам потрібно розробити структуру майбутньої довідки. Вміст довідки являє собою деревоподібну структуру, у якій можуть бути **книги** й **сторінки**. Книга в HTML Help Workshop - це розділ, що має підрозділи, такі книги можуть бути вкладеними. Сторінка - це кінцевий розділ, що не має підрозділів. Наша програма не настільки складна, тому й структура буде досить простій:

Призначення програми
Команди меню
    Файл
    Виправлення
    Формат
    Кодування
    Довідка
Зворотний зв’язок

Тут ми маємо дві сторінки: «**Призначення програми**» і «**Зворотний зв’язок**», а також книгу «**Команди меню**», що у свою чергу, містить ще 5 сторінок: «**Файл**», «**Виправлення**», «**Формат**», «**Кодування**» і «**Довідка**».

Таким чином, нам оведеться зробити 8 HTML-файлів, окремо на книгу й на кожну зі сторінок. Я не буду приводити повні листинги цих файлів - умист довідки буде залежати від вашої фантазії, але деякі показові теги я приведу. От, наприклад, як я вказував посилання на існуючі сторінки керівництва:

```
<body>
В программе CodeBook имеются следующие пункты меню:<br>
<ul>
  <li><a href="File.html">Файл</a></li>
  <li><a href="Pravka.html">Правка</a></li>
  <li><a href="Format.html">Формат</a></li>
  <li><a href="Coder.html">Кодирование</a></li>
  <li><a href="Spravka.html">Справка</a></li>
</ul>
</body>
```

Рис. 28.2. Демонстрація посилань на інші сторінки керівництва

Код наведено у вигляді зображення, щоб теги приклада не конфліктували з тегами самої лекції. Причому текст HTML-файлу відкритий у простому Блокноті, щоб побачити теги в NVU, потрібно відкрити вкладку «**Код**» у нижній частині робочого вікна.

А так у мене вставляється зображення:

```
<br>
<img style="width: 540px; height: 307px;" alt="Вид редактора с зашифрованным текстом"
 title="Зашифрованный текст" src="MyText.JPG"><br>
<span style="font-style: italic;"><br>
```

Рис. 28.3. Демонстрація посилань на зображення

У результаті повинна вийти приблизно така сторінка із вставленим зображенням:

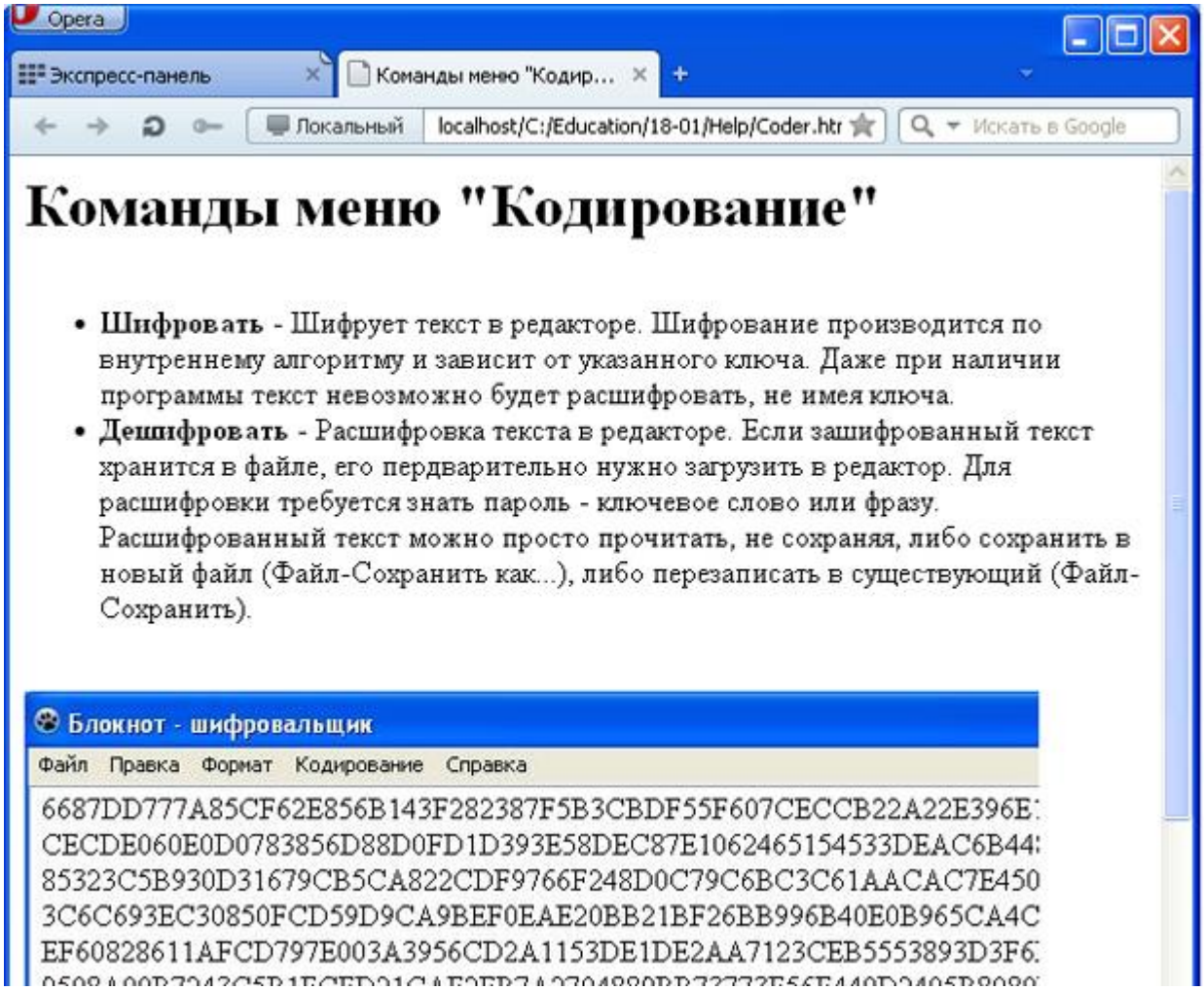


Рис. 28.4. Зовнішній вигляд сторінки із зображенням

Сподіваюся, зміст ви вловили, і правильно зробили всі 8 html-файлів (у мене є ще 9-й файл - зображення **MyText.jpg**).

Наш **Блокнот-шифрувальник** перебуває в папці **18-01**, якщо ви додержувалися моїх рекомендацій. Створимо в цій папці ще одну папку **Help**, де будемо збирати нашу довідкову систему. У мене вийшла така адреса:

C:\Education\18-01\Help

Тут має бути розміщено 8 html-файлів:

- Coder.html
- CommandsMenu.html
- File.html
- Format.html
- Naznachenie.html
- ObratSvyaz.html

- Pravka.html
- Spravka.html

і картинка MyText.JPG, щовставлена у сторінці Coder.html (див. рис. 28.4).

Тепер вся підготовка зроблена, можна завантажувати програму HTML Help Workshop і зайнятися, нарешті, створенням CHM-файлу. Завантаживши програму, виберіть команду меню «**File -> New**», або клацніть кнопку «**New**» на **Панелі керування**. Перед вами з'явиться наступне віконце:

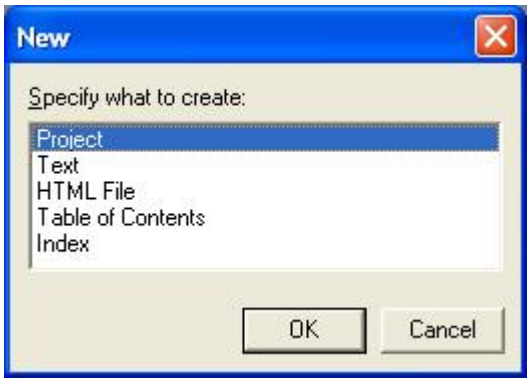


Рис. 28.5. Вікно вибору об'єкта в HTML Help Workshop

Отже, тут ви можете вибрати для створення наступні об'єкти:

- **Project** - Проєкт у цілому. Це нам вибирати ще рано, спочатку потрібно буде зробити Таблицю Змісту.
- **Text** - Простий текстовий файл.
- **HTML File** - Файл HTML, веб-сторінка. Однак, як уже говорилося, можливості цього редактора досить скромні, і щоб ним користуватися, потрібні знання гіпертекстової розмітки.
- **Table of Contents** - Таблиця Змісту, файл формату **HHC** (Hyper Help Contents). Тут відкриється редактор, у якому потрібно буде ввести зміст контекстної довідки, точно така ж, як розроблена нами раніше деревоподібна структура. І потрібно буде зв'язати кожну книгу й сторінку зі своїм HTML-файлом.
- **Index** - Створення індексного файлу, або файлу покажчиків. У цьому файлі збираються ключові слова, фрази, за якими потім будується індекс. Зараз в Інтернеті такі ключові слова нерідко називають тегами. Файл має формат HHK (Hyper Help Keyword).

Створення Таблиці змісту

Як говорилося вище, спочатку потрібно буде зробити **Таблицю змісту**, тому вибирайте об'єкт **Table of Contents**.

У вікні **HTML Help Workshop** буде відкритий об’єкт **TOC** (Table Of Contents):



Рис. 28.6. Таблиця змісту в HTML Help Workshop

Значущими для нас є деякі кнопки на лівій бічній панелі інструментів. Сама верхня кнопка - **Властивості таблиці змісту**. Дві наступні кнопки дозволяють додати в таблицю книгу або сторінку, це те, що нам зараз знадобиться.

Натисніть кнопку «**Insert a page**» (назви кнопок з’являються, якщо якийсь час потримати над ними курсор миші) для додавання сторінки. У поле «**Entry title**» потрібно вказати ту назва розділу, що буде відображатися в таблиці. У нас це буде «**Призначення програми**» (без лапок, звичайно). Потім потрібно нажати кнопку «**Add/Edit**», щоб вказати відповідний розділ HTML-файл. Відкриється ще одне вікно «**Path or URL**», у якому потрібно ввести ім’я HTML-файлу або його Інтернет-адреса. Можна скористатися кнопкою «**Browse**» для вибору файлу, але тоді файл буде зазначений разом з його відносною адресою, наприклад, так:

..\..\Education\18-01\Help\Naznachenie.html

У нас із вами всі файли проєкту довідника будуть перебувати в одній папці, тому відносна адреса отут не потрібна. Видаліть її, залишивши

в рядку «**File or URL**» тільки ім'я відповідному розділу HTML-файлу, у мене це:

Naznachenie.html

Натисніть кнопку «**ОК**», і тоді вікно додавання роздягнула **Таблиці змісту** буде виглядати так:

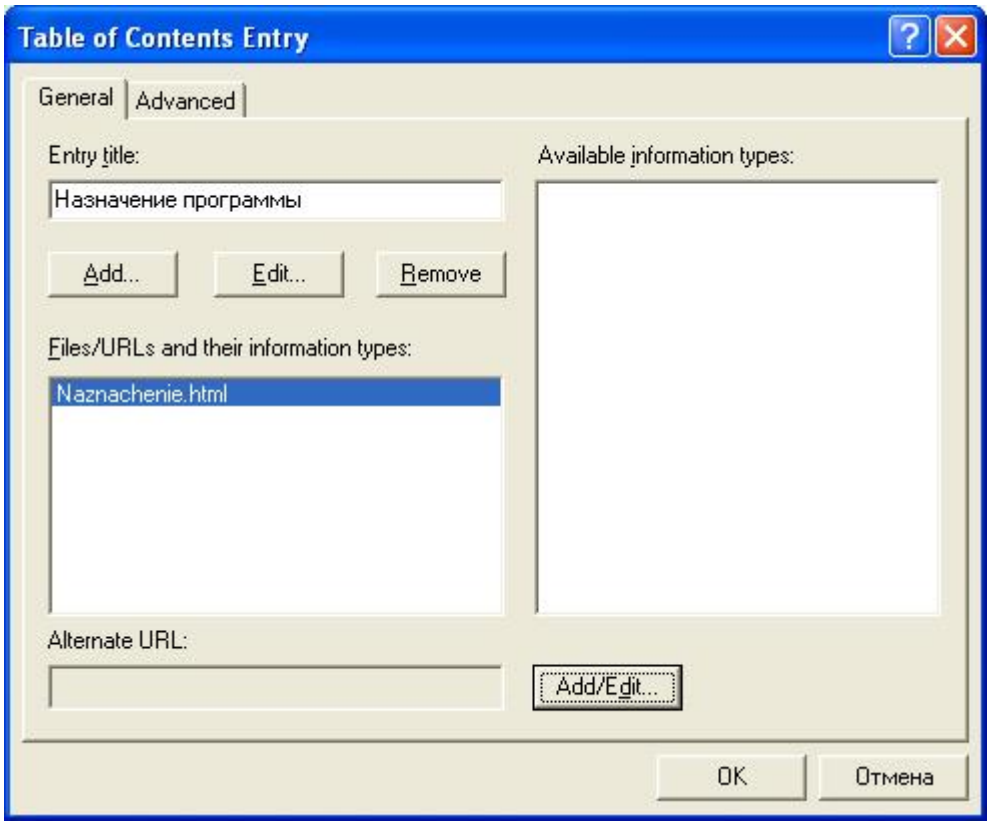


Рис. 28.7. Додавання роздягнула в Таблицю змісту

Натисніть «**ОК**», і розділ буде доданий у **Таблицю змісту**.

Тепер натисніть кнопку «**Insert a heading**», тому що цього разу нам потрібно додати не сторінку, а книгу «**Команди меню**». Якщо ви впевнені ви, що бажаєте додати книгу в таблицю змісту, відповідайте ствердно. Процес додавання книги абсолютно такий же, як і сторінки. Вкажіть назву розділу «**Команди меню**» і вкажіть HTML-файл, що відповідає цьому розділу. Не забудьте видалити відносну адресу, якщо ви скористалися кнопкою «**Browse**» для вибору файлу.

Потім ви вкажете усе, що потрібно, і підтвердите всі запити, книга буде додана в **Таблицю змісту**, але зявиться ВИЩЕ попередньої сторінки! А в нашій деревоподібній структурі вона повинна бути в другому рядку. Отут рятуйте! чотири кнопки зі стрілками. Перші дві дозволяють переміщувати об'єкт вище й нижче, а другі - уліво й вправо. Таким чином, ми можемо не тільки встановити розділ на

потрібний рядок **Таблиці**, але й пересунути його на потрібну позицію деревоподібної структури. Адже в третьому рядку в нас повинна бути сторінка «**Файл**», що буде вкладеною в книгу «**Команди меню**», тобто, повинна бути зміщена вправо.

Проробивши цю операцію потрібна кількість разів, додайте всі інші сторінки в **Таблицю змісту**:

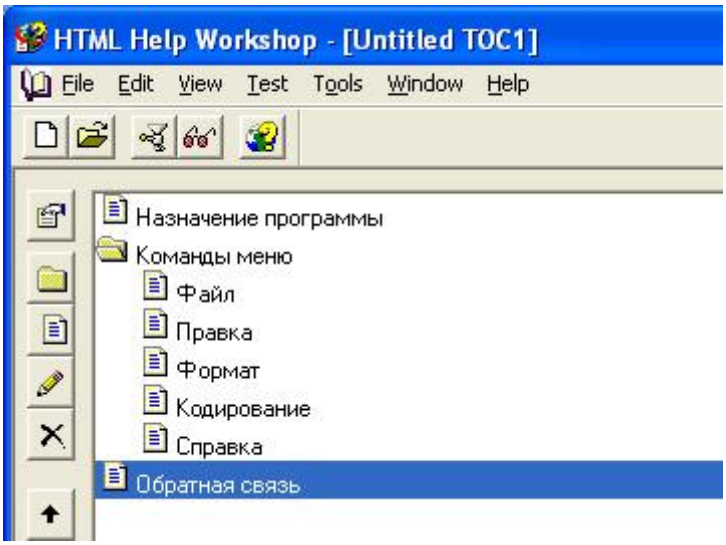


Рис. 28.8. Розділи Таблиці змісту

Не забувайте забирати відносну адресу HTML-сторінок.

Коли закінчите роботу, натисніть нижню кнопку «**Save file**» на лівій панелі інструментів для збереження **Таблиці змісту**. Ім'я таблиці можете залишити без зміни, «**Table of Contents.hhc**», але переконаєтеся, що зберігаєте його в ту ж папку, де вже перебувають HTML-файли довідки. Втім, ця сама папка вже повинна бути зазначена у вікні збереження **Таблиці**, якщо ви додавали HTML-сторінки кнопкою «**Browse**».

2. Створення файлу проєкту

Командою головного меню «**File -> Close all**» закрийте створену раніше **Таблицю розділів**. Потім виберіть команду «**File -> New**» або натисніть кнопку «**New**» на **Панелі інструментів**, щоб почати створення нового об'єкта. Цього разу ми створюємо **Проєкт**, тому у вікні вибору об'єкта вкажіть **Project**.

Появится вікно майстра створення проєкту, і спочатку вам буде запропоновано конвертувати існуючий проєкт застарілого формату **WinHelp**. Ми з вами створюємо проєкт «з нуля», тому прапорець «**Convert WinHelp project**» не встановлюємо, а відразу натискаємо «**Далі**».

На наступному кроці вам буде запропоновано ввести ім’я файлу проєкту або вибрати існуючий файл. Отут найпростіше нажати кнопку «**Browse**», переконатися, що відкрито папку «**Help**», у якій у нас зберігаються всі файли проєкту довідки, у поле «**Ім’я файлу**» указати ім’я файлу проєкту. Отут є одне зауваження: результуюча довідка буде мати таке ж ім’я, як і файл проєкту, а імена довідкових файлів звичайно збігаються з іменами програми. Файл програми в нас називається **CodeBook.exe**, виходить, вписуємо ім’я **CodeBook** і натискаємо «**Відкрити**». Ви повернетеся у вікно з ім’ям файлу, це ім’я буде зазначено разом з адресою, а ім’я файлу проєкту автоматично одержить розширення HHP (Hyper Help Project). У мене вийшло так:

C:\Education\18-01\Help\CodeBook.hhp

Натискаємо «**Далі**», і відкриється наступне вікно майстра. Отут нам буде запропоновано вказати, які файли проєкту ми вже розташовуємо. У нас із вами є HTML-файли з розділами й файл **Таблиці змісту**, встановлюємо відповідні прапорці:



Рис. 28.9. Вікно майстра створення проєкту - вказівка існуючих об’єктів

Натискаємо «**Далі**». На наступному кроці буде запропоновано вказати файл **Таблиці змісту**. Натисніть «**Browse**» і виберіть файл таблиці (**Table of Contents.hhc**). Натискаємо «**Відкрити**», адреса й ім’я файлу **Таблиці** потрапить у поле майстра. Натискаємо «**Далі**».

На наступному кроці нам пропонується ввести всі HTML-файли проєкту. Можна додавати їх по черзі, а краще всі файли разом. Натисніть кнопку «**Add**», відкриється вікно відкриття файлів. За замовчуванням, повинна бути відкрита папка «**Help**». У вікні ви побачите 8 HTML-файлів. Виділите перший з них, потім, утримуючи кнопку **<Shift>**, клацніть по останньому. Всі 8 файлів виявляться

виділеними. Натисніть «**Відкрити**», і всі ці файли потраплять у вікно майстра:

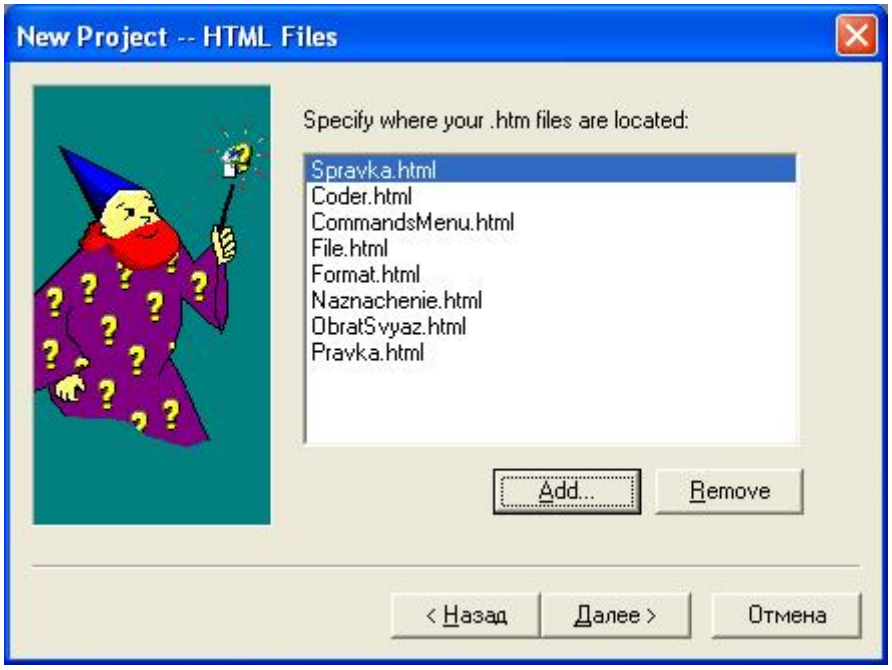


Рис. 28.10. Вікно майстра створення проєкту - вказівка існуючих HTML-файлів

Натискаємо кнопку «**Далі**», потім «**Готове**». Все, проєкт створений.

Створення індексів

Наша довідка майже готова, ми вже можемо скомпілювати результуючий файл довідки, однак не заважає додати в неї деякі додаткові функції. Індексу те в нас ще немає!

Індекс - це файл із розширенням **HHK** (Hyper Help Keyword), що містить ключові слова й фрази, і посилання на сторінки довідки, до яких вони ставляться.

Зараз в Інтернеті багато сайтів - бібліотек, блогов, новостных сайтів і т.п. - де ви можете знайти інформацію з так називаних тегів. Ви ж бачили поле, куди вводять шукані слова й фрази, і кнопку «**Знайти**» поруч? Ті ключові слова, які ви туди вводите, у сфері Інтернету називаються тегами, а в проєкті довідки - індексами.

Одна сторінка довідки може мати безліч ключових слів. З іншого боку, одне ключове слово може посилатися на безліч сторінок довідки. Грамотна організація індексів сильно полегшить кінцевому користувачеві роботу з довідкою, отут все залежить від вашої фантазії й знань психології користувача.

У вікні **HTML Help Workshop** відкритий файл проєкту, у лівій частині вікна ви бачите три вкладки: «**Project**», «**Contents**» і «**Index**».

Щоб почати створення індексів, потрібно перейти на вкладку «**Index**». Оскільки індексний файл у проєкті ще не зазначений, то буде виведений запит - чи створюємо ми новий індексний файл, або підключаємо існуючий? Ми створюємо новий файл, тому залишіть виділеної радиокнопку «**Create a new index file**» і натисніть «**ОК**». Вам буде запропоновано ввести ім'я індексного файлу, причому за замовчуванням буде запропоноване ім'я **Index.hhk**. Не бачу особливого змісту миняти це ім'я. Переконаєтеся тільки, що в поле «**Папка**» як і раніше відкрита наша папка «**Help**», індексний файл нам потрібно зберігати теж у неї. Натисніть «**Зберегти**», і індексний файл буде створений.

Пока ще він порожній - індексів немає. Займемося їхнім створенням. Взагалі, варто продумати заздалегідь, які ключові слова до якого розділу довідки прив'язати. У мене вийшло так:

Таблиця 28.1. Зразковий список ключових слів і фраз для індексного файлу

Файл і назва роздягнула	**Ключові слова**
Coder.html Команди меню «**Кодування**»	кодування, декодування, шифрувати, дешифрувати, розшифрувати, текст
CommandsMenu.html Команди меню	пункти меню, файл, виправлення, формат, кодування, довідка
File.html Команди меню «**Файл**»	файл, робота з файлом, створити, відкрити, зберегти, статистика, вихід
Format.html Команди меню «**Формат**»	формат, шрифт, кольори, перенос за словами, автоматичний перенос тексту
Naznachenie.html Призначення програми	CodeBook, блокнот-шифрувальник, блокнот, пароль, ключове слово, версія
ObratSvyaz.html Зворотний зв'язок	зворотний зв'язок, сайт, e-mail, електронний ящик, автор
Pravka.html Команди меню «**Виправлення**»	виправлення, скасувати, вирізати, копіювати, вставити, видалити, виділити всі, буфер обмину

У вас імена файлів і назви розділів можуть відрізнятися. Крім того, можливо, ваша фантазія підкаже, як розширити список ключових слів.

Зі списком розібралися, приступаємо до створення індексів. У вікні **HTML Help Workshop** повинна бути відкрита вкладка «**Index**». Зверніть увагу на кнопки лівої бічної панелі інструментів, вони майже такі ж, як у **Таблиці змісту**. Нас цікавить друга кнопка «**Insert a keyword**», на ній зображений ключ.

Натисніть на неї. Відкриється вікно додавання індексу. Ключове слово або фразу потрібно вписати в поле «**Keyword**», потім нажати кнопку «**Add**». Буде відкрите вікно, у якому потрібно вибрати розділ, на який буде посилатися дане ключове слово. Якщо це слово зустрічається в різних розділах, то прийде створювати кілька індексів: з одним ключовим словом, але різними посиланнями на розділи. Для більших довідкових систем та ще робота!

Отже, ключове слово вказали, розділ довідки в поле «**HTML titles**» вибрали - поля «**File or URL**» і «**Title**» повинні заповнитися автоматично. Натисніть «**ОК**», і новий індекс з'явиться у вікні додавання індексу:

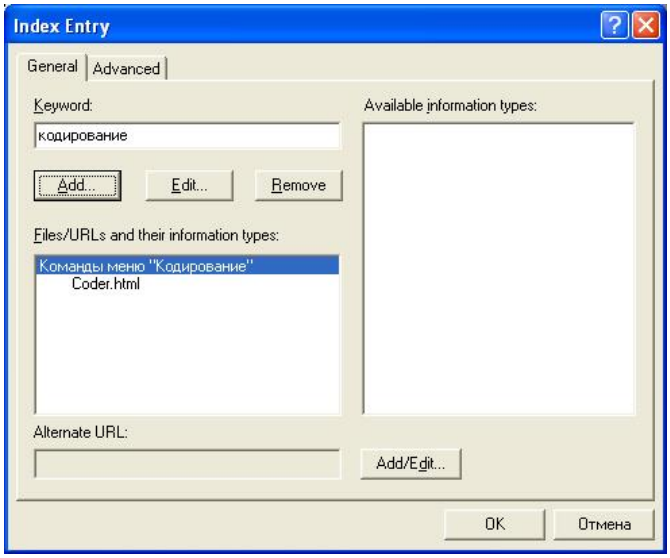


Рис. 28.11. Вікно додавання ключового слова

Ще раз натисніть «**ОК**», і на вкладці «**Index**» з'явиться ключове слово «**кодування**». Нічого складного, крім того, що подібну операцію прийде повторювати багаторазово, для кожної пари «**Ключове слово - Посилання на розділ**». Так що запасіться терпінням, і введіть всі ваші ключові слова й фрази.

Щораз вам буде виводитися запит - чи варто створювати новий індекс на верху списку - відповідайте ствердно, ми цей список відсортуємо потім. Якщо ви випадково помилилися, можете нажати третю зверху кнопку із зображенням олівця для редагування індексу,

або четверту - для видалення, тільки цей помилковий індекс повинен бути виділений.

Не знаю, як у вас, а в мене вийшло 46 ключових слів і фраз! Не заважало б розсортувати їх за абеткою. Для цього служить третя знизу кнопка «**Sort keywords alphabetically**» з буквами A і Z і стрілкою вниз - натисніть її, і список буде відсортований. Якщо вам доведется працювати з великою довідкою, індекси можна створювати не за один підхід, а додавати їхній час від часу. Але потім все-таки не забувайте їх знову сортувати, інакше індекси будуть виглядати неохайно.

Всі, роботу з індексами закінчили, натисніть нижню кнопку «**Save file**» для збереження індексного файлу.

3. Створення повнотекстового пошуку

Індекси можуть істотно полегшити користувачеві пошук потрібної довідки, однак все-таки **повнотекстовий пошук** для цього використають частіше. Повнотекстовий пошук - це коли користувач уписує шукане слово/фразу, а довідкова система шукає це слово/фразу по всій довідці. Індекси при цьому не використаються, тобто, знайти можна будь-яке слово, не тільки проіндексоване.

За зручність використання повнотекстового пошуку доводиться платити деяким уповільненням пошуку, однак на сучасних ПК це вповільнення майже непомитно.

Включити повнотекстовий пошук дуже просто. У вікні **HTML Help Workshop** відкрийте вкладку «**Project**». Сама верхня кнопка лівої бічної панелі інструментів - «**Change project options**» (Змінити параметри проєкту). Натисніть її, перейдіть на внеску «**Compiler**» і включите прапорець «**Compile full-text search information**»:

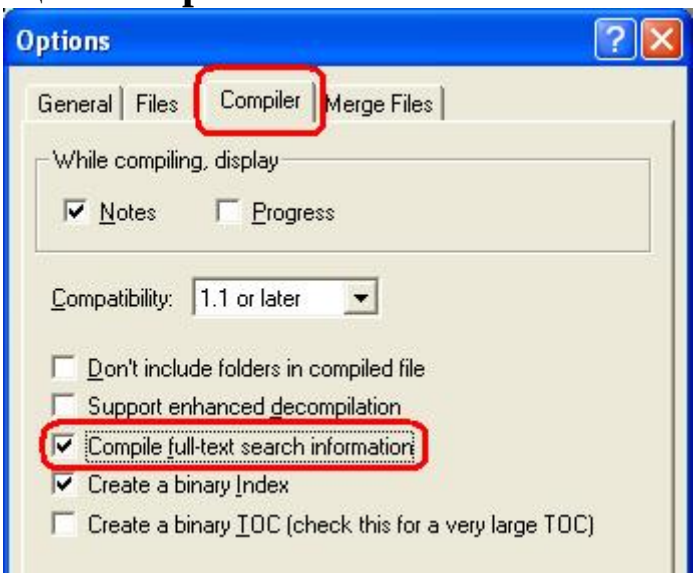


Рис. 28.12. Вікно зміни параметрів проєкту

Крім того, варто прямо зараз указати, яка сторінка довідки буде відкриватися за замовчуванням. Для цього перейдіть на вкладку «**General**», і в поле «**Default file**» виберіть потрібний файл. Я, наприклад, вибрав **Naznachenie.html**. Не забудьте нажати кнопку «**ОК**», щоб підтвердити зміни.

Створення вкладки «Обране»

Вкладка «**Обране**» у довідковій системи може послужити гідним завершальним штрихом. У цю вкладку користувач буде мати можливість додавати ті розділи довідки, до яких найчастіше звертається. Створюється ця вкладка теж досить просто.

На вкладці «**Project**» HTML Help Workshop, на лівій бічній панелі інструментів, третя кнопка зверху - кнопка «**Add/Modify window definitions**» (Додати/Змінити визначення вікна). Довідка адже виводиться у вікні, ми своє вікно поки не створювали, користувалися тим, що створюється за замовчуванням, а тому до настроювань вікна доступу не маємо. От нам і потрібно створити своє вікно довідки. Натисніть на цю кнопку. Вам буде запропоновано ввести ім’я нового вікна. Ім’я може бути кожне, наприклад, **Favorites**.

Як тільки ви введете ім’я й натиснете «**ОК**», відкриється вікно **Window Types** з настроюваннями нашого нового вікна. Для додавання вкладки «**Обране**» нам потрібно перейти на вкладку «**Navigation Pane**» і включити прапорець «**Favorites tab**»:

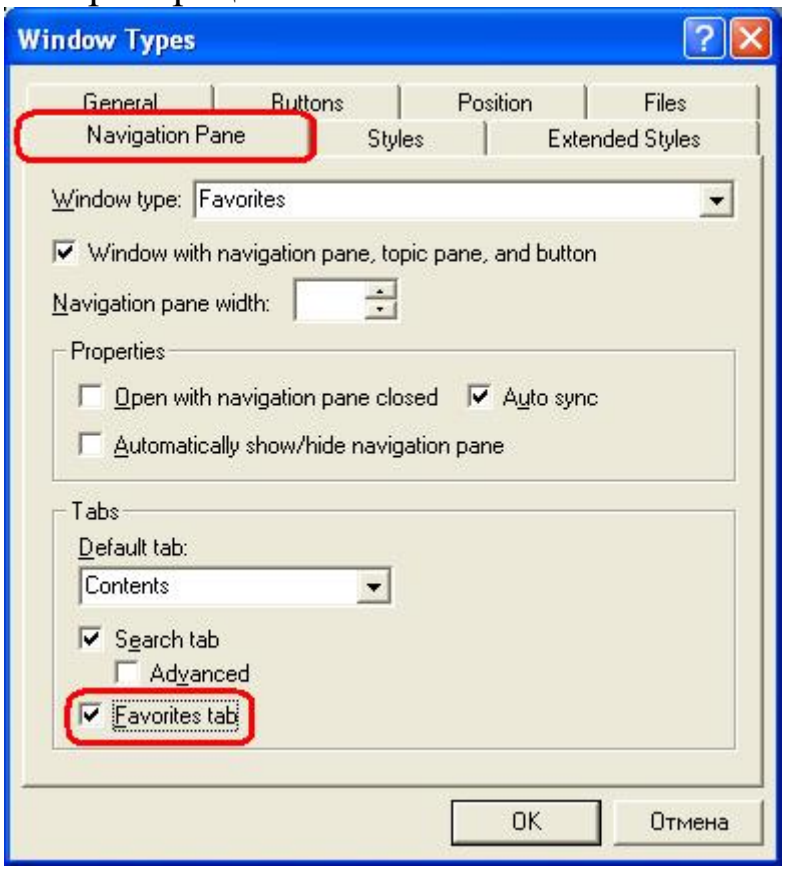


Рис. 28.13. Включення вкладки Обране

Крім того, перейдіть на вкладку «**Files**» і в поле «**Default**» також укажіть файл довідки, що буде відкриватися за замовчуванням.

Натисніть «**ОК**», і на цьому наша робота із проєктом завершена. Не забудьте зберегти проєкт командою меню «**File -> Save project**» або кнопкою із зображенням дискетки.

Компіляція довідки

Наш проєкт закінчений, залишилося скомпілювати саму довідку - файл **CodeBook.chm**. Робиться це дуже просто: виберіть команду головного меню «**File -> Compile…**»... Вам буде запропоновано вибрати файл проєкту, для якого компілюється довідка, за замовчуванням буде відкритий наш проєкт

C:\Education\18-01\Help\CodeBook.hhp

Це нам і потрібно. Рекомендую включити прапорець «**Save all files before compiling**» (Зберегти всі файли перед компіляцією). Після чого можете нажати кнопку «**Compile**».

Якщо ви всі зробили правильно, у правій частині вікна вийде звіт про пророблену компіляцію:

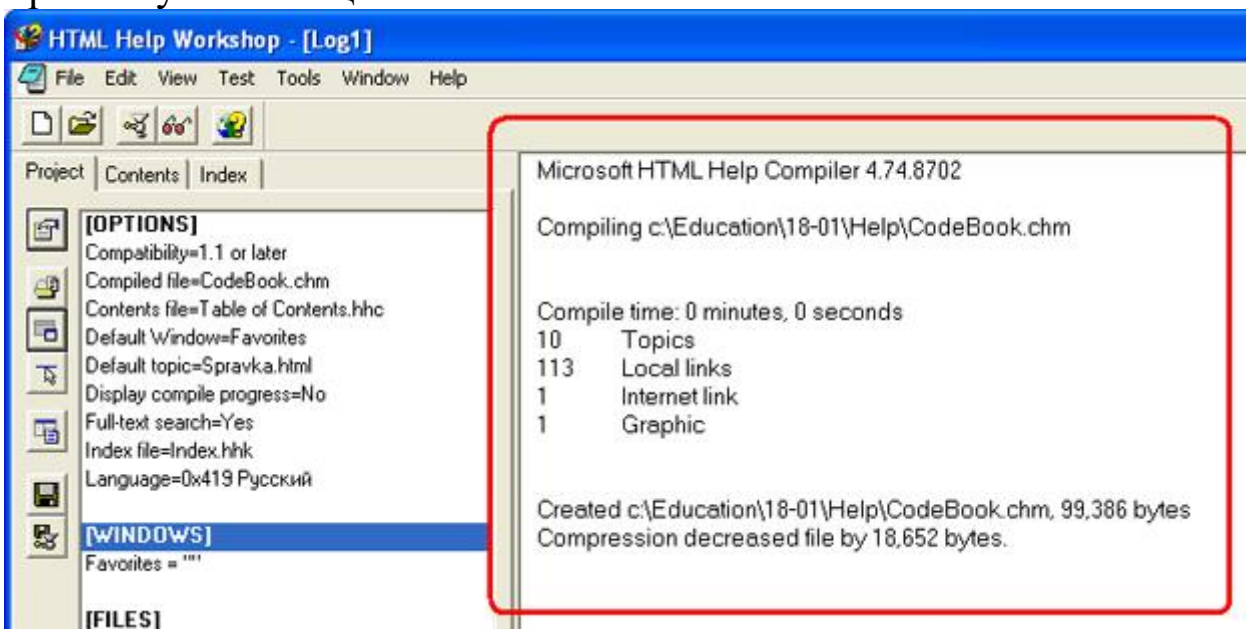


Рис. 28.14. Звіт про компіляції

Як бачите, помилок у звіті немає, зазначена кількість тим, локальних і Заслань-інтернет-посилань, зображень. Правда, не завжди ці кількості відповідають дійсності, але отут уже нічого не поробити.

Також зазначено, що був створений файл **CodeBook.chm** розміром 99,386 байт. На цьому робота із проєктом завершена, можете закрити **HTML Help Workshop**.

Потім можете Провідником або файловим менеджером відкрити папку **Help** із проєктом довідки, і запустити файл **CodeBook.chm** прямо звідтіля:

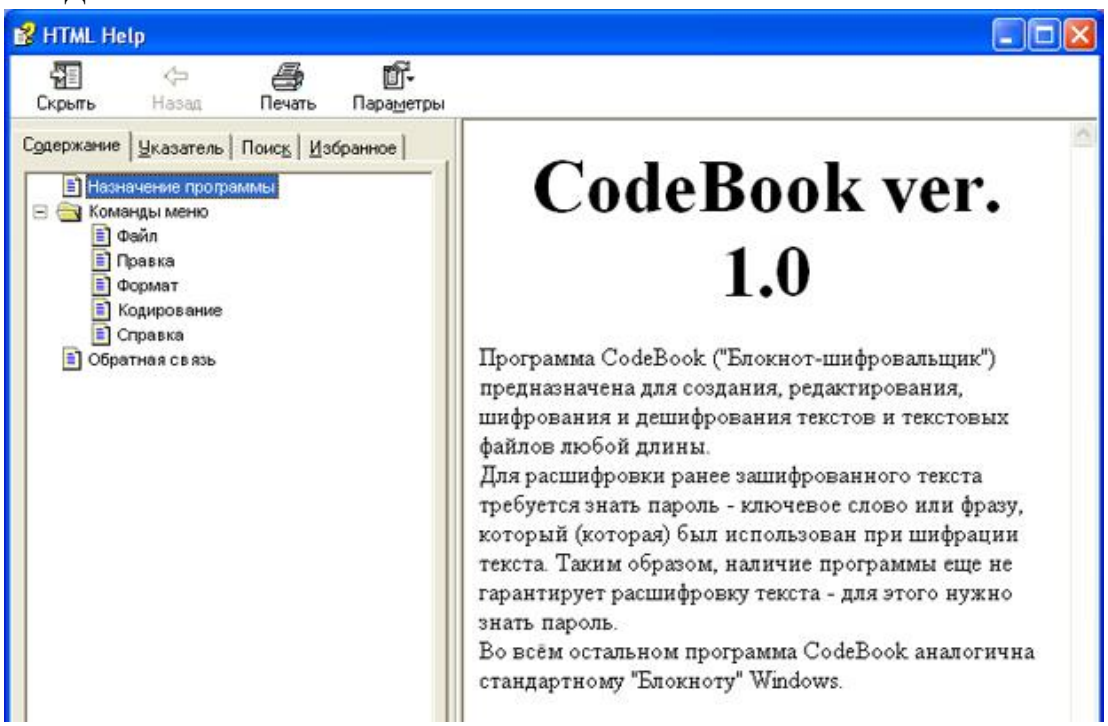


Рис. 28.15. Працююча довідкова система

Як бачите, довідка відкривається, на ній присутні всі 4 вкладки: «**Зміст**», «**Покажчик**», «**Пошук**» і «**Обране**». Можете поэкспериментировать із цими вкладками.

Є декілька зауважень. Перше: для показу цієї довідки система задіє інструменти програми MS Internet Explorer, але на кожній Windows ця програма встановлена, так що турбуватися нема чого. Друге: довідка самостійно запам’ятовує останнє положення й розміри, які встановив користувач. При наступному запуску довідки ці положення й розміри будуть відновлені, що сильно спростить нам роботу.

Включення довідки в проєкт Lazarus

Треба сказати, механізм підключення довідкової системи до проєкту в Lazarus ще вогкуватий, і інформації із цього питання небагато. Одні радять використати кілька десятків рядків коду, інші говорять, що це не допомагає…В офіційній документації (англійською мовою) Lazarus радять використати компоненти THTMLHelpDatabase і THTMLBrowserHelpViewer із вкладки **System**. Як приклад пропонують проєкт

C:\lazarus\examples\helphtml

Цей приклад, чомусь виводить довідку в браузері, встановленому в системі за замовчуванням. Потім мені все-таки вдалося запустити довідку, однак довідка відкрилася у вікні перегляду Lazarus, а не системи. Це вікно, як виявилося, з кирилицею «не дружить» - замість російських букв виходили прямоугольнички.

Зрештою, мені вдалося знайти досить простий спосіб підключити до проєкту довідкову систему, про нього я вам зараз і розповім.

Насамперед, скопіюйте файл довідки **CodeBook.chm** у папку із проєктом програми. Можна було б залишити його й у папці **Help**, особливої різниці, звідки запускати довідник немає. Однак коли всі файли програми перебувають в одній папці, те цю програму простіше підготувати до поширення.

Далі, відкрийте проєкт програми в Lazarus - запустіть файл **CodeBook.lpi** з папки із проєктом, у мене це **C:\Education\18-01**. Відкриється **Lazarus** із проєктом програми. У **Редакторі коду** переконаєтеся, що відкрито вкладку «**Main**» - вкладка головного вікна. Перейдіть на редактор форм. Відкрийте редактор головного меню (двічі клацніть по компоненті MainMenu1).

Нам потрібно додати ще один елемент у розділ меню «**Довідка**». Виділите цей розділ, виділите його елемент «**Про програму**», клацніть правою кнопкою миші й виберіть команду «**Вставити новий пункт (перед)**». З'явиться новий пункт меню. Виділите його, властивість Name перейменуйте в HelpHelp, у властивості Caption напишіть Довідка, а у властивості ShortCut виберіть гарячу клавішу **<F1>**.

Згенеруйте для цього пункту меню подія OnClick, код якого буде досить простій:

```
procedure TfMain.HelpHelpClick(Sender: TObject);
begin
  //викликаємо довідку:
  WinExec('hh.exe CodeBook.chm',SW_SHOW);
end;
```

Процедура WinExec викликає зовнішній системний додаток **hh.exe** (системний переглядач довідок) із завантаженим файлом довідки **CodeBook.chm**. Однак **WinExec** відразу ж працювати не буде - ця процедура прописана в модулі Windows, якому потрібно додати в розділ uses. Додайте, збережете проєкт, перекомпілюйте його й запустіть. Натисніть **<F1>** або виберіть команду меню «**Довідка -> Довідка**» - наша довідка повинна завантажитися.

Якщо ви захочете поширювати цю програму, вам для цього буде досить двох файлів: **CodeBook.exe** (сама програма) і **CodeBook.chm** (довідка).

Тема 29. Створення інсталятора

1. Навіщо потрібні інсталятори
2. Огляд інсталяторів
3. Файл ліцензії
4. Створення інсталятора в Inno Setup

1. Навіщо потрібні інсталятори

Наш курс підходить до кінцю. Ми написали безліч проєктів, навчилися робити довідку й підключати її до програми, але хотілося б ще якісь із проєктів опублікувати в Мережі! А що для цього потрібно? Настановний файл. До речі, якщо ви будете писати програми на замовлення, то створення інсталятора майже напевно буде обов'язковою умовою проєкту.

Здавалося б, можна обійтися яким-небудь популярним архіватором начебто WinRAR, зробити на ньому SFX-архів, що саморозпаковується, прописати, який файл програми куди повинен бути розпакований. Однак так ви одержите лише примитивний інсталятор, у якому користувачеві не надається ніяких можливостей вибору, а це погано - навіть якщо користувач при установці програм звичайно використає всі настроювання «за замовчуванням», натискаючи тільки кнопки «**Далі**» або «**Next**», йому подобається усвідомлювати, що від нього щось залежить, що він має можливість вибору.

Припустімо, ваш проєкт досить складний, містить десятки файлів. Ви бажаєте, щоб користувач мав вибір способу установки вашої програми: куди встановлювати, потрібно чи створювати ярлики на робочому столі й у панелі швидкого запуску? Чи встановлювати повну версію, вибіркову або мінімальну? Чи реєструвати програму в реєстрі або створити INI-файл? Крім того, користувач може коли-небудь захотіти цю програму видалити зі свого ПК, а виходить, потрібно передбачити й зворотний процес - деінсталяцію. Всі ці операції зробити у звичайному архіваторе буде неможливо, або, принаймні, надзвичайно складно. Зате ці ж операції легко створюються в

спеціальних програмах-інсталяторах, які входять у групу утиліт програміста. Про їх і будемо говорити на цій лекції.

2. Огляд інсталяторів

Деякі середовища розробки програм мають у своєму складі інсталятори. Так, до складу Delphi входить безкоштовний інсталятор Install Shield Express, що, до речі, можна скачати із сайту виробника. Уважається, що якщо до складу IDE входить інсталятор, то для створення інсталяційного файлу (або CD) краще користуватися ім. Install Shield Express, наприклад, має дуже зручну функцію включення в інсталяційний пакет «движка» баз даних BDE, що, втім, уже застарів, і тепер практично не використається. Нічого іншого видатного в Install Shield Express не спостерігаються, тому навіть Delphi-програмісти предпочитают більше «просунуті» інсталятори.

Lazarus взагалі не має у своєму складі подібної утиліти, однак існує безліч інсталяторів сторонніх розроблювачів - як платних, так і безкоштовних, за допомогою яких Lazarus-програміст може створити інсталяційний файл або навіть завантажувальний CD/DVD.

Які ще інсталятори існують? Їхня безліч: Setup Factory, Visual Studio Installer, Nullsoft Install System, Smart Install Maker і багато інших.

Особливо хотілося б виділити один із самих популярних безкоштовних інсталяторів, що володіють широкими можливостями - Inno Setup. Це система створення інсталяторів для Windows-програм. Inno Setup з’явився в 1997 році і є не просто безкоштовним, але й відкритим проєктом (Open Source), перевершує багато комерційних установників по функціональності й стабільності. Ми будемо розглядати створення інсталяторів на основі Inno Setup.

Inno Setup можна скачати із сайту виробника, офіційна сторінка: http://www.jrsoftware.org/isinfo.php

Або ж, ви можете скачати цю програму з мого сайту:

Варто помитити, що при написанні курсу розглядалася версія Inno Setup 5.5.4, а вони досить часто обновляються. Так що на сайті виробника, можливо, ви знайдете більше свіжу версію.

3. Файл ліцензії

Звичайно при установці якої або програми, користувачеві спочатку пропонується прочитати й прийняти ліцензійну угоду (ліцензію) програми. Користувач читає (або частіше, не читає) ліцензію, установлює радиокнопку з написом «**Я приймаю умови**» (або щось аналогічне), після чого стає доступної кнопка «**Далі**». Напевно, наш

інсталятор повинен також виводити яку те ліцензію? Де неї взяти, або як її зробити, не будучи юристом?

Отут рада простій: знайдіть схожу з вашої програму, і використайте текст її ліцензії, переробивши лише назву програми, копирайта (того, кому належить авторське право), номер версії, і т.п. Тобто, якщо ви збираєтеся опублікувати в Інтернеті вашу програму, як безкоштовну (freeware), те й ліцензію копіюйте в іншої безкоштовної програми. Оскільки всі ліцензії мають приблизно однакові пропоновані умови й використають однакову терминологію, то нічого страшного в даній хитрості немає. Створіть звичайний текстовий файл із текстом ліцензії, наприклад, у стандартному Блокноті. Ми розглянемо створення інсталятора для нашого Блокнота-шифрувальника з лабораторної роботи теми № 18, тому й файл із ліцензійною угодою повинен бути про цю програму. Назвіть файл **License.txt** (так звичайно називають файли з ліцензією), а текст цього файлу може бути приблизно таким:

«CodeBook» (Блокнот-шифрувальник)

Copyright (c) 2014, В.Ю. Ачкасов

Версія 1.0 (freeware)

-i-i-i-i-i-i-i-i-i-i-i-i-i-i-i-i-i-i-i-i-i-i-i-i-i-i-i

До установки й/або використання цього програмного забезпечення, уважно ознайомтеся з умовами цієї ліцензійної угоди й обмеженої гарантії.

ЛІЦЕНЗІЙНА УГОДА. АВТОРСЬКЕ ПРАВО

Програмне забезпечення «CodeBook», включаючи документацію й інші додаткові матеріали, захищено законами й мижнародними угодами про авторські права, а також іншими законами й договорами, що регулюють відносини авторського права. Дане програмне забезпечення ліцензується, а не продається.

Всіма авторськими правами на програму «CodeBook» володіє тільки її автор - В’ячеслав Юрійович Ачкасов. Усі права, не надані тут явно, зберігаються за автором. Автор залишає за собою право скасувати дію даної ліцензії в кожній з наступних версій програми.

ВИКОРИСТАННЯ Й ПОШИРЕННЯ:

Поточна версія «CodeBook», документація й супутні файли можуть вільно використатися, а також копіюватися, передаватися й поширюватися будь-якими способами при дотриманні наступних умов:

- Файли повинні поширюватися в незминному вихідному виді. Це повинен бути вихідний архів, сформований автором. Не допускається

стягнення плати за поширення «CodeBook» без письмового дозволу автора.
«CodeBook» не може бути проданий або перепроданий, включений до складу іншого програмного забезпечення, використатися в службі підтримки комерційного підприємства, або використатися для іншого способу одержання прибутку, без письмового дозволу автора.
- При цитуванні, передруці або іншім використанні будь-яких фрагментів документації або файлів, що входять в «CodeBook», посилання на автора й джерело цитування обов’язкові.
- Ви не можете використати, копіювати, эмулировать, створювати нові версії, здавати внайми або оренду, продавати, зминювати, передавати програму або будь-які з її складових, інакше, чим визначене дійсною ліцензійною угодою.
- Ви не можете декомпілювати, дизасемблювати, вивчати код програми іншими способами, крім випадків, коли така діяльність явно дозволяється законом, незважаючи на це обмеження.
- Будь-яке таке нелегальне використання означає автоматичне й негайне припинення дії даної угоди й може переслідуватися за законом.

ГАРАНТІЇ

«CodeBook» поширюється за принципом «ЯК Є». Будь-які наслідки, фінансові й інші втрати, що можуть наступити внаслідок використання «CodeBook» самої по собі й/або разом з іншим програмним забезпеченням не можуть бути приводом для пред’явлення яких-небудь претензій до автора й офіційно вповноваженим розповсюджувачам.

Установка й/або використання «CodeBook» означає, що ви розумієте положення дійсної ліцензійної угоди й згодні з ними. Якщо або ви не згодні із цією ліцензійною угодою, ви не маєте права встановлювати або використати дане програмне забезпечення й вам необхідно видалити файли дистрибутива «CodeBook» з ваших пристроїв зберігання інформації й припинити їхнє використання.

Copyright (c) 2021, В.Ю. Ачкасов

Зрозуміло, ви можете змінити цей текст - додати або видалити які або розділи, змінити прізвище правообладателя на свою, додати власні

дані для зворотного зв’язку (e-mail, ваш сайт). Але в цілому, текст файлу з ліцензійною угодою повинен виглядати приблизно так.

4. Створення інсталятора в Inno Setup

Ми розглянемо найпростіший випадок створення інсталятора. Буквально, у кілька кличів миші. Однак можливості Inno Setup куди ширше. При необхідності, ви освоїте їх самостійно за допомогою досить докладної довідки (правда, англійською мовою, хоча в Інтернеті можна знайти й русифіковані варіанти).

Отже, для створення інсталятора нам насамперед буде потрібно створити папку з файлами програми, нехай це буде

C:\CodeBook

У цю папку нам потрібно буде скопіювати три файли:

1. **CodeBook.exe** - Виконує файл, що, програми
2. **CodeBook.chm** - Файл довідкової системи
3. **License.txt** - Файл із ліцензійною угодою

Звичайно, кількість і склад файлів залежить від проєкту - деякі проєкти можуть мати десятки, а те й сотні файлів. Нам же для даного проєкту досить буде цих трьох файлів.

Завантажуємо Inno Setup. Виходить наступне вікно вітання:

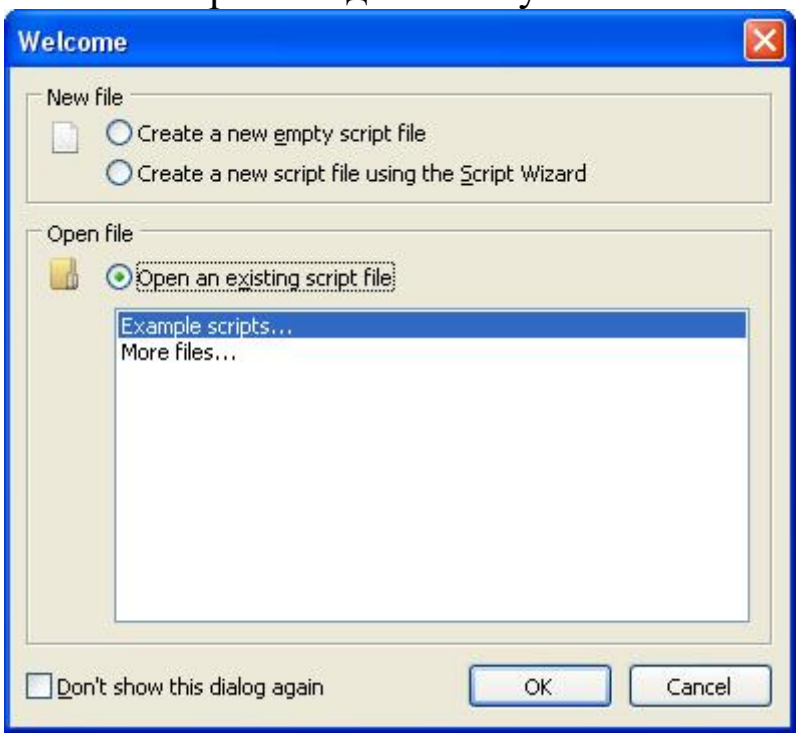


Рис. 29.1. Початкове вікно Inno Setup

Тут ми можемо вибрати наступні пункти:

У секції «**New file**»:

- **Create a new empty script file** - Створення нового порожнього скриптового файлу. Справа в тому, що кожний проєкт створення інсталятора в Inno Setup - це набір скриптов, внутрішніх команд, які описують сценарій створення інсталятора й використовувані настроювання. Якщо ви вивчите ці скрипты, то зможете створювати інсталятори будь-якої складності. Але на даному етапі нам це не потрібно.
- **Create a new script file using the Script Wizard** - Створення нового скриптового файлу з використанням Майстра Сценаріїв. Це те, що нам зараз потрібно, включите цю радиокнопку.

Є ще розділ «**Open file**» з пунктом **Open an existing script file** - Відкрити й використати існуючий файл сценаріїв. Цей пункт може вам придатися, якщо ви захочете змінити існуючий сценарій.

Отже, включаємо **Create a new script file using the Script Wizard** і натискаємо «**ОК**». На наступному кроці буде виведене вітання Майстра сценаріїв, отут просто натискаємо кнопку «**Next**». Прапорець **Create a new empty script file** включати не потрібно.

Далі вийде вікно з основною інформацією про додаток:

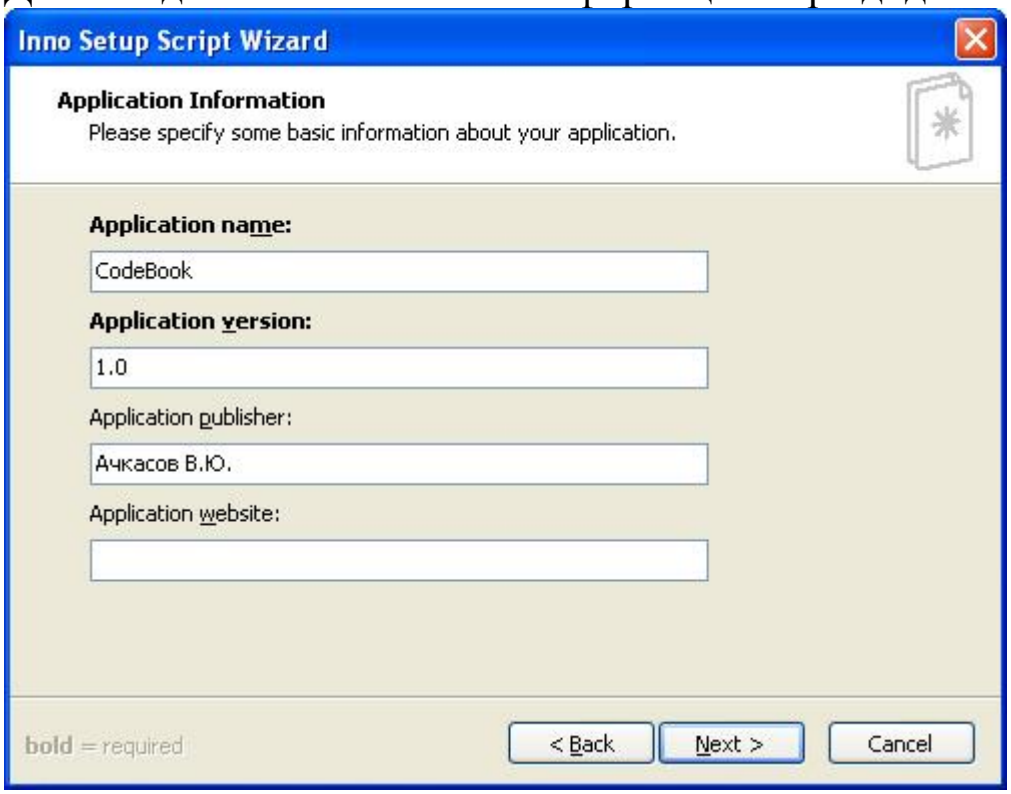


Рис. 29.2. Вікно основних налаштувань

Тут нам потрібно вказати наступні налаштування:

- **Application name** - Назва додатка, у нашому випадку CodeBook.

- **Application version** - Версія додатка, у нашому випадку 1.0.
- **Application publisher** - Кому належить авторське право, отут можете вказати своє Прізвище, Ім'я та По батькові.
- **Application website** - Офіційний сайт програми. Якщо такого сайту немає, залишаємо поле порожнім.

Натискаємо «**Next**». Далі буде виведене вікно папки проєкту:

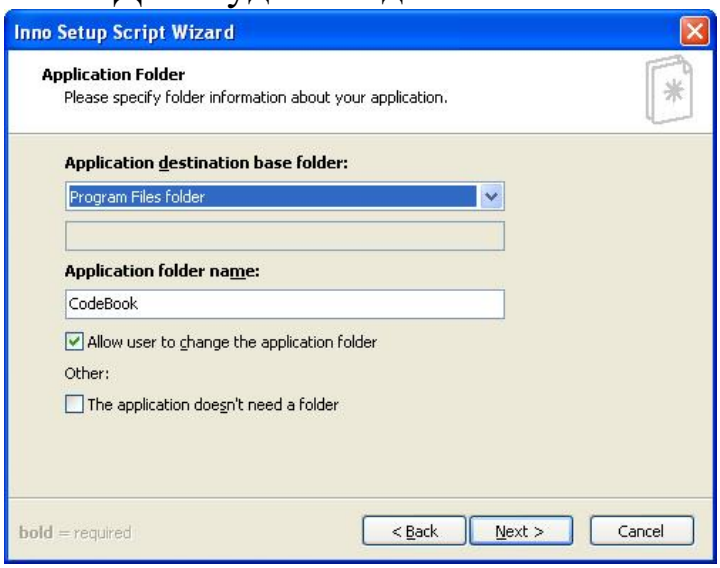


Рис. 29.3. Вікно папки проєкту

У поле «**Application destination base folder**» ми повинні вказати, куди за замовчуванням буде встановлюватися наша програма. За замовчуванням, це папка **C:\Program Files**. Так і залишимо: якщо користувач укаже інший шлях, то програма встановиться туди, інакше - в **C:\Program Files**.

У поле «**Application folder name**» потрібно вказати папку, куди буде скопійована програма. У нашому випадку виходить, що повною адресою установки програми буде **C:\Program Files\CodeBook**.

Прапорець «**Allow user to change the application folder**» у включеному положенні дозволить користувачеві змінити папку установки програми, запропоновану за замовчуванням. Правила гарного тону цього вимагають, так що залишаємо прапорець включеним.

Прапорець «**The application doesn't need a folder**» (Цей додаток не має потреби в папці) означає, що програми не потрібна власна папка, виходить, вона буде встановлена за адресою **C:\Windows**. Це іноді роблять для системних утиліт, але наша програма в папці бідує, так що прапорець повинен бути виключений.

Натискаємо «**Next**». Далі буде виведене вікно, у якому буде запропоновано вказати головний файл програми, а також додати інші файли (і папки), які повинні бути встановлені:

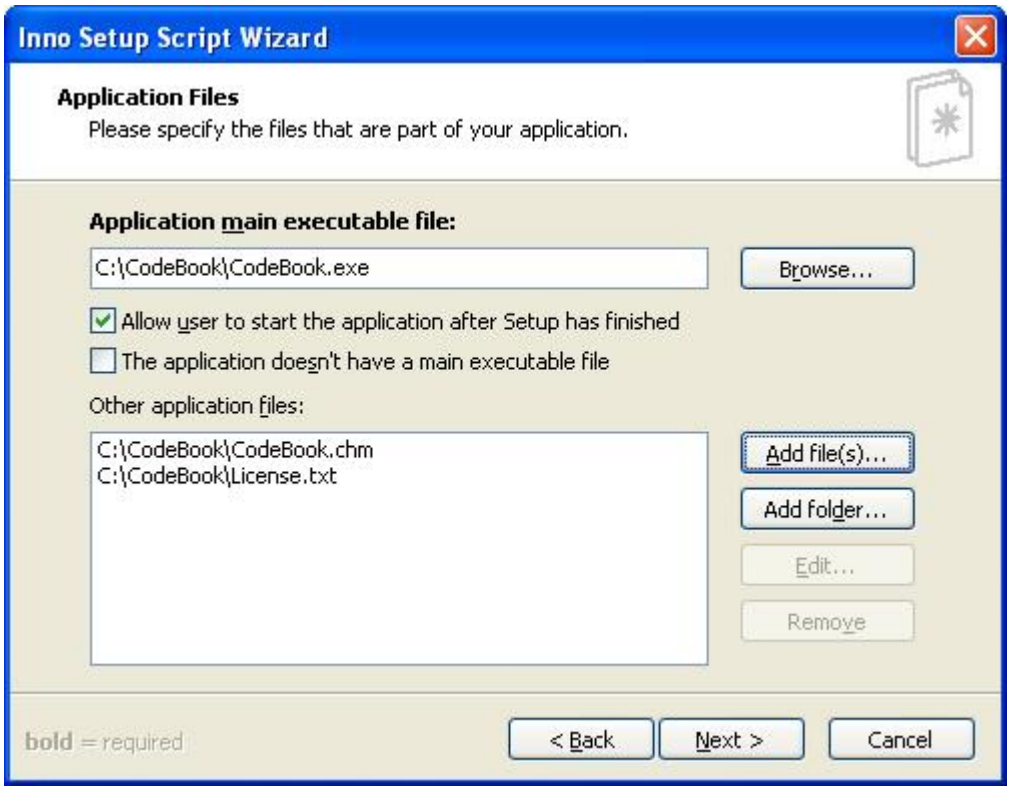


Рис. 29.4. Вказівка файлів (і папок) проєкту

У поле «**Application main executable file**» потрібно вказати головний файл програми. Натисніть кнопку «**Browse**» і знайдіть цей файл, як на малюнку (всі файли створюваного інсталятора ми помистили в папку **C:\CodeBook**, якщо ви помнете).

Кнопка «**Add file(s)**» дозволяє додати інші необхідні файли. Нам потрібно додати файл довідки (обов'язково), і файл ліцензії (якщо бажаємо, щоб інсталятор його теж скопіював при установці), як на малюнку.

Кнопка «**Add folder**» дозволяє додавати вкладені папки в папку програми. У нас таких папок ні, так що цього ми не робимо.

Прапорець «**Allow user to start the application after Setup has finished**» (Дозволити користувачеві запуск додатка після установки) залишаємо включеним, у цьому випадку, після установки вийде запит про запуск додатка.

А от прапорець «**The application doesn't have a main executable file**» (У додатка немає головного файлу, що виконує,) нам включати не потрібно.

Натискаємо «**Next**». Далі буде виведене вікно папки стартового меню:

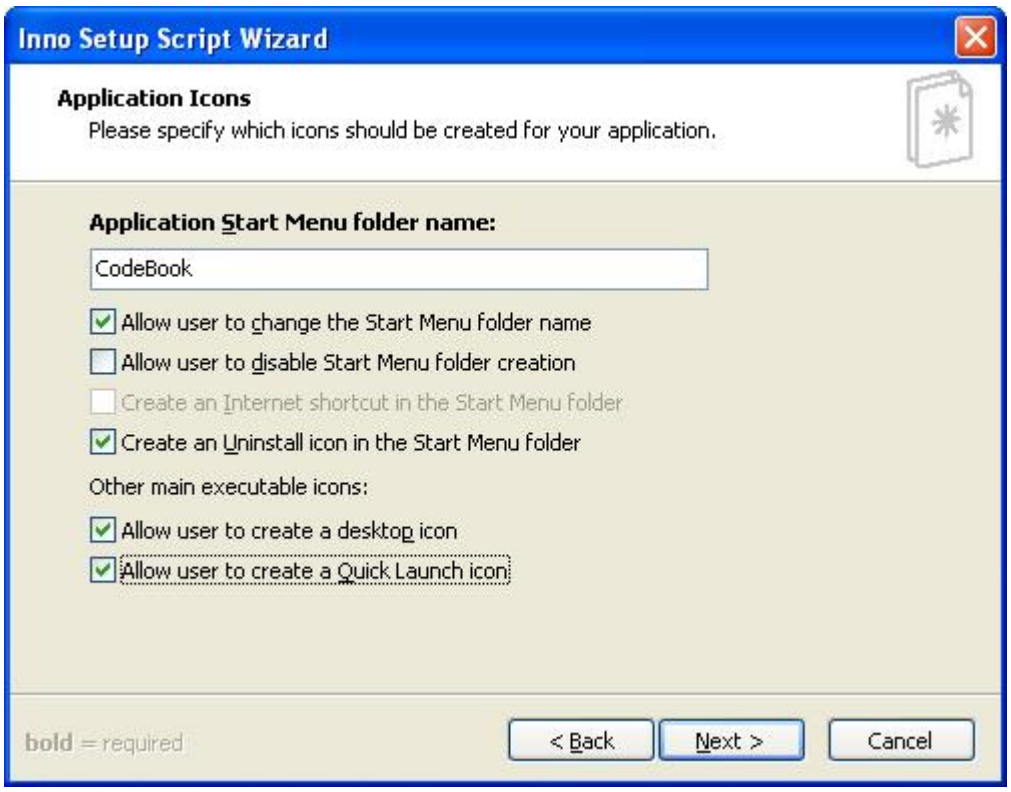


Рис. 29.5. Вікно папки стартового меню програми

Тут, у поле «**Application Start Menu folder name**» (Ім'я папки стартового меню програми) потрібно ввести ім'я папки стартового меню, у якій будуть розташовуватися ярлики програми. Звичайна назва цієї папки збігається з назвою програми, але при бажанні, можете його змінити.

Прапорець «**Allow user to change the Start Menu folder name**» (Дозволити користувачеві змінити ім'я папки **Стартового Меню**) краще залишити включеним, так вимагають правила гарного тону.

Прапорець «**Allow user to disable Start Menu folder creation**» (Дозволити користувачеві відключити створення папки **Стартового Меню**) включати не потрібно - ми адже хочемо, щоб папка **CodeBook** була присутня в **Стартовому Меню**!

Прапорець «**Create an Internet shortcut in the Start Menu folder**» при включеному стані додає в папку **Стартового меню** посилання на офіційний сайт програми, якщо такий існує. Якщо ви помнете, ми не вказували сайт програми, тому даний прапорець не доступний. Однак якщо офіційний сайт програми існує, і ми його вказали, то даний прапорець потрібно буде включити, це приведе до того, що в папку **Стартового меню** із програмою буде додане посилання на сайт програми.

Прапорець «**Create an Uninstall icon in the Start Menu Folder**» при включеному стані створить команду деінсталяції програми. Правила гарного тону просто вимагають, щоб даний прапорець був ВКЛЮЧЕНИЙ!

Останні два прапорці дозволяють користувачеві вибрати, чи будуть створюватися ярлики програми на **Робочому столі**, і на панелі швидкого запуску. Обидва прапорці бажано включити, так користувачеві буде дано більше волі вибору.

Натискаємо «**Next**». Далі буде запропоновано вказати файл(ы) з різною інформацією:

У поле «**License file**» пропонується вказати файл із ліцензією, вона буде виведена у вікні інсталятора в процесі установки програми. Натисніть «**Browse**» і виберіть наш файл ліцензії, як на малюнку.

У поле «**Information file shown before installation**» пропонується вказати текстовий файл із інформацією, що буде виведена у вікні інсталятора ПЕРЕД установкою. Якщо у вас є ради про установку, які бажано вивести користувачеві, можна оформити їх у вигляді текстового файлу й указати цей файл у даному полі. У нас такої інформації немає (так звичайно її взагалі не вказують), залишаємо поле порожнім.

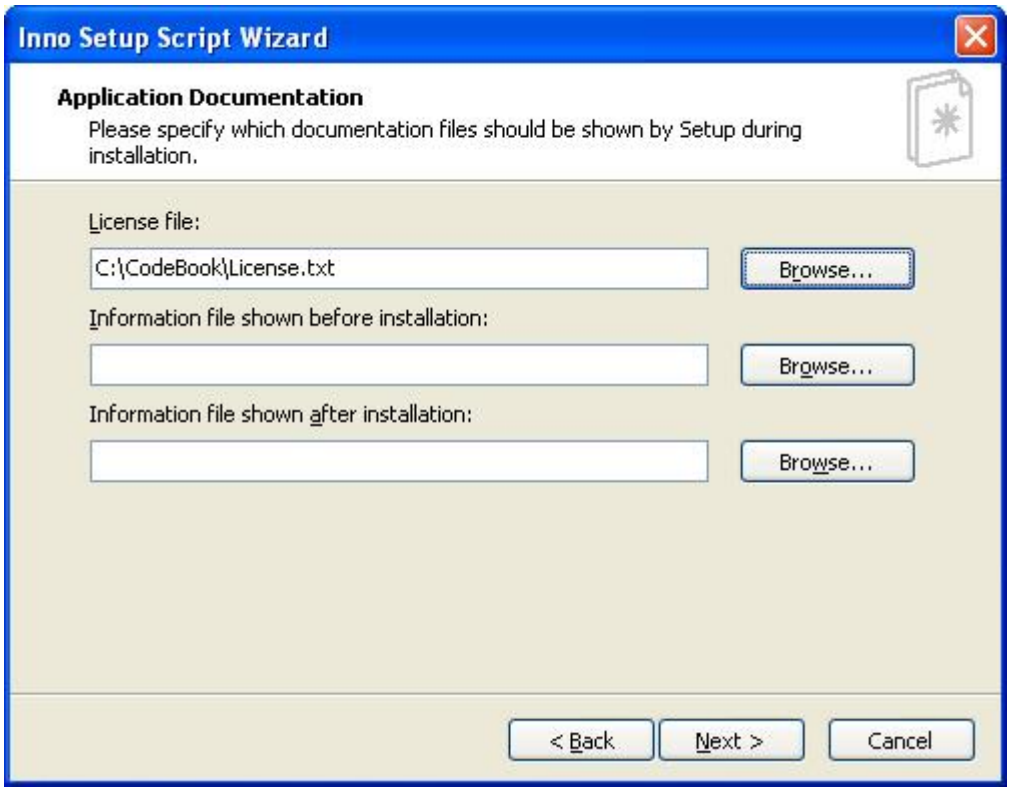


Рис. 29.6. Вікно вибору файлів з додатковою інформацією

У поле «**Information file shown after installation**» пропонується вказати текстовий файл із інформацією, що буде виведена у вікні інсталятора ПІСЛЯ установки. Такої інформації в нас також немає.

Натискаємо «**Next**». Нам будуть запропоновані вибрати мови установки:



Рис. 29.7. Вікно вибору мов установки

Список можливих мов вражає. З огляду на, що в користувача може бути англійська версія Windows, залишаємо включеним англійську мову, і, звичайно ж, включаємо росіянин. Майте на увазі, що це мова інсталятора програми, а не мова інтерфейсу самої програми! Перше вікно інсталятора в кожному разі вийде мовою операційної системи. Потім користувач може вибрати іншу мову, і наступні вікна інсталятора буде вже на ньому.

Снову натискаємо «**Next**». У наступному вікні, у полі «**Custom compiler output folder**» нам буде потрібно вказати папку, у яку буде помищений результат компіляції проєкту - властиво, сам інсталяційний файл. Укажемо папку

C:\OutCodeBook (папка вже повинна існувати):

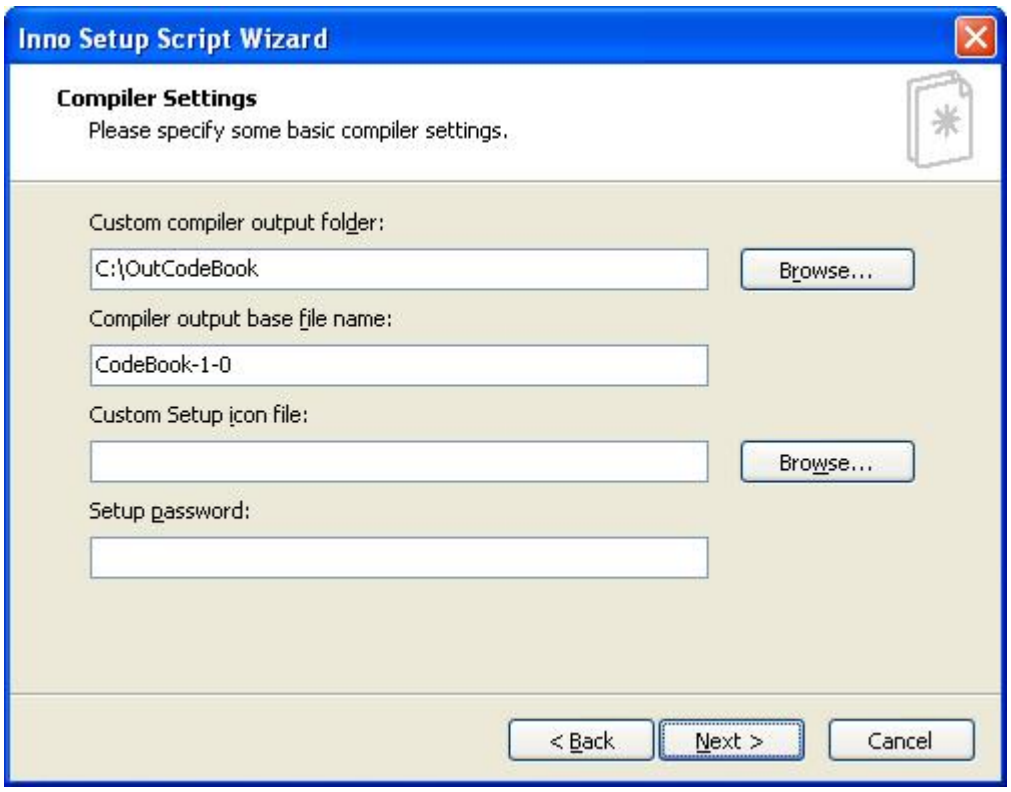


Рис. 29.8. Вікно кінцевих параметрів установки

У поле «**Compiler output base file name**» потрібно буде ввести ім'я інсталяційного файлу нашої програми. Можна залишити його за замовчуванням - **setup**, можна змінити на **install**, але зараз всі частіше інсталяційний файл містить назву й версію програми (як це прийнято в Unix-подібних ОС). У мене це ім'я **CodeBook-1-0**. Розширення *.exe буде додано автоматично, отут його вказувати не потрібно.

Якщо у вас є графічний ICO-файл із «іконкою», що ви бажаєте застосувати до вашого додатка, то можете вибрати цей файл у поле «**Custom Setup icon file**». Якщо ви це зробите, то нова «іконка» програми буде відображатися й у папці **Стартового Меню**, і на **Робочому столі**, і в панелі **Швидкого запуску**. У нас такого файлу ні, ми його не вказуємо, виходить, буде використана «іконка» за замовчуванням - стандартна іконка Lazarus-програм - зображення синього відбитка лабети гепарда.

І, нарешті, у поле «**Setup password**» можна вказати пароль запуску установки. Хто не знає пароль, не зможе встановити програму. Цю можливість можна використати для створення платних програм - хто заплатив за програму, тому висилаєте пароль, щоб він змиг установити її. Однак не варто покладати на цю можливість більших надій. Швидше за все, перший же оплатив програму користувач викладе цей пароль в Інтернеті, і інших оплат ви не дочекаєтеся. Крім того,

концепція shareware має на увазі, що ви даєте користувачеві пробний період, наприклад, 15 днів, після якого він або повинен заплатити за програму, або видалити її із ПК. Не знаючи пароля, користувач не зможе й випробувати програму. Так що не вказуємо ніякого пароля (тим більше, що наша програма має ліцензію freeware, тобто, безкоштовна), і натискаємо «**Next**».

У наступн вікн запитува_, потрібно чи включа у проєкт #define-директивы. Це свого роду, константні імена команд. Залишаємо прапорець «**Yes, use #define compiler directives**» включеним, і натискаємо «**Next**» востаннє.

Вийде завершальне вікно майстра. Натиснемо кнопку «**Finish**», і проєкт інсталятора буде створений. Однак це ще тільки проєкт, а не сам інсталятор. Після роботи майстра вийде запит:



Рис. 29.9. Завершальний запит: потрібно чи компілювати новий скрипт зараз

Натисніть «**Так**». Далі вийде запит, потрібно чи зберігати скрипт проєкту перед компіляцією? Скрипт проєкту - це звичайний текстовий файл із розширенням ISS, у якому зберігається вся та інформація, що ми вводили в майстру. Якщо є ймовірність, що ви будете цей проєкт компілювати ще раз, має сенс нажати «**Так**» і зберегти скрипт.

Далі буде запропоновано ввести ім’я скрипта, назвемо його, як і програму - **CodeBook**. За замовчуванням, буде відкрита папка з іншими файлами програми. Після збереження скрипта відбудеться компіляція проєкту, вікно **Inno Setup** буде виглядати так:

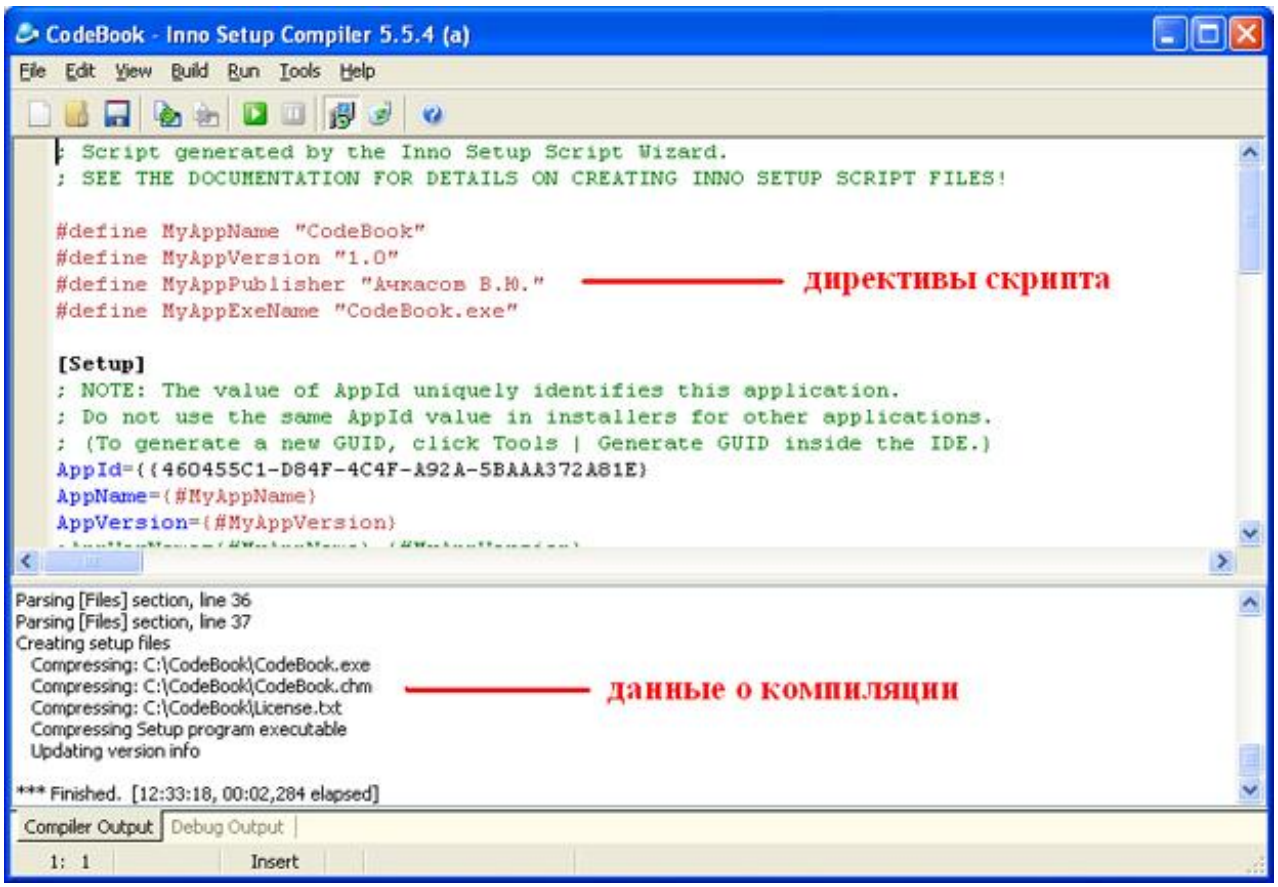


Рис. 29.10. Вікно Inno Setup після компіляції

У верхній частині розташовані директиви скрипта - сценарій створення інсталятора. У нижньої - інформація про компіляції. Тепер, якщо ви заглянете у вихідну папку (**C:\OutCodeBook**), що вказували раніше, те виявите там файл інсталятора **CodeBook-1-0.exe**. Причому, якщо три файли проєкту займають 1,9 Мб, те файл інсталяції буде займати всього те менш 900 Кб. Стиск вийшов більш ніж удвічі!

Тепер можете запустити інсталятор і протестувати його роботу: Встановіть програму, подивитеся, що перебуває в папці **C:\Program Files\CodeBook** (якщо не зминювали папку установки), подивитеся, чи створилася папка **CodeBook** у **Стартовому меню**, які там команди (повинні бути дві команди: «**CodeBook**» і «**Деінсталювати CodeBook**»). Видалите програму, спробуйте встановити її на іншій мові.

Примитка: програма для створення інсталяторів Inno Setup настільки популярна, що для неї сторонніми розроблювачами створюються різні розширення. Так, можу порадити розширення **Inno Script Studio**, що є надбудовою над Inno Setup. Inno Script Studio має більше зрозумилий інтерфейс (причому є можливість установки російського інтерфейсу), спрощену наступну роботу з директивами скрипта.

Наш проєкт **CodeBook** остаточно завершений, тепер можете дарувати його друзям і однокласникам, розмищати в Інтернеті, застосовувати отримані знання в інших проєктах. Сподіваюся, даний курс був вам корисний. Удачі в програмуванні!

Список використаних джерел та корисні посилання

1. http://lazarus.freepascal.org/ - Офіційний сайт Lazarus. Тут ви зможете й нові версії знайти, і новини почитати.
2. http://wiki.freepascal.org/Main_Page/ua - документація по Lazarus і Free Pascal. Якщо постаратися, отут можна знайти відповіді майже на всі питання, які неодмінно у вас будуть виникати.
3. http://www.cyberforum.ua/lazarus/ - Великий форум по Lazarus. Форуми взагалі дуже корисні й інформативні, а форуми про програмування особливо. Навіть якщо ви й не змогли знайти в Інтернеті рішення вашої проблеми, на форумі неодмінно знайдеться який-небудь «гуру», що вам допоможе порадою.
4. http://www.freepascal.ua/ - інформаційний портал про Free Pascal і Lazarus. Теж досить корисний сайт.