C++ Builder 4.0

C++ Builder 4.0 je dvoucestný vizuální nástroj pro vývoj aplikací v jazyce C++. Je silný; spojuje v sobě přednosti vizuálních vývojových prostředí a jazyka C++. Umožňuje vyvíjet graficky orientované aplikace pro Win32 tak i konzolové aplikace, distribuované databázové aplikace, aplikace pro WWW atd.

Co v této knize najdete

Tato kniha vám nabízí nenásilný úvod do programování v C++ Builderu. Na řadě příkladů se seznámíte s nejdůležitějšími komponentami z knihovny VCL, která tvoří základ C++ Builderu. V první kapitole se prostě jen rozhlédneme po prostředí a seznámíme se se základy jeho ovládání. Ve druhé kapitole najdete volné variace na „Hello, world“, program, kterým začínají dnes snad všechny učebnice programovacích jazyků. Přitom se seznámíte s řadou užitečných komponent, naučíte se vytvářet dialogová okna atd.

Ve třetí kapitole napíšeme program, který ukazuje přesný čas. To nám umožní seznámit se mj. s časovačem, s principy kreslení v C++ Builderu a s používáním tzv. prostředků (resources). Ve čtvrté kapitole vytvoříme prohlížeč grafických souborů. Přitom se mj. naučíme používat standardní dialogová okna, pracovat s bitmapovými soubory a vytvářet aplikace s rozhraním MDI. V páté kapitole vytvoříme jednoduchý vektorový grafický editor. To nám poskytne záminku k podrobnějšímu seznámení s používáním grafických nástrojů a s některými třídami ze standardní šablonové knihovny. Mimo to se naučíme, jak z programu tisknout.V šesté kapitole napíšeme program, který kreslí jakési složité obrázky — fraktály. Přitom se naučíme, jak vytvořit program pro Windows, který se „chová slušně“, i když provádí složité a dlouhé výpočty. Dále se mj. seznámíme se základy programování vícevláknových (multithreadových aplikací), naučíme se vytvářet a používat dynamické knihovny, pracovat s informacemi o verzi programu atd.

V sedmé kapitole nás čeká první seznámení s databázemi. Poznáme význam základních komponent, naučíme se přistupovat k databázovým tabulkám, používat uspořádání master--detail a využívat filtry. V osmé kapitole zajdeme o kousek dále — naučíme se vytvářet základní dotazy v jazyce SQL a využívat je v C++ Builderu. Naučíme se také procházet jednotlivé záznamy, aktualizovat je atd.

V deváté kapitole se velice stručně setkáme s programováním pro internet a v desáté kapitole napíšeme svoji první vizuální komponentu.

V posledních dvou kapitolách najdete přehled rozšíření jazyka C++ v C++ Builderu, přehled významu nastavení vývojového prostředí a přehled postupů při nejběžnějších operacích.

Co v této knize nenajdete

Především zde nenajdete učebnici jazyků C a C++ ani učebnici programování pro Windows pomocí Windows API. Nenajdete zde ani podrobný a vyčerpávající popis prostředí nebo referenční přehled knihovny VCL, tedy vizuálních komponent.

Nehledejte tu také výklad o obecných principech tvorby podnikových databázových aplikací (klient/server nebo vícevrstvých), neboť to přesahuje rámec knihy. Také podrobný výklad o jazyku SQL by vydal na samostatnou knihu, a nijak tenkou.

Nedostalo se ani na vytváření komponent ActiveX nebo na programování distribuovaných aplikací podle standardu CORBA nebo COM.

Co byste měli znát

V této knize předpokládám, že umíte pracovat s PC vybaveným operačním systémem Windows 95/98 nebo Windows NT. Vedle toho předpokládám, že znáte programovací jazyk C a základy jazyka C++. Přesněji, že umíte deklarovat objektový typ, umíte zacházet s dědičností a znáte základy používání výjimek v C++ a tušíte, co to jsou prostory jmen.

Doprovodné CD

Na doprovodném CD najdete zdrojové texty všech příkladů z této knihy, ve dvou verzích, pro C++ Builder 3 a pro C++ Builder 4. Vedle zdrojových souborů, souborů s prostředky a projektových souborů zde najdete i spustitelné soubory (EXE). Tyto programy lze ovšem spustit jen v prostředí, ve kterém jsou instalovány i potřebné dynamické knihovny (VCL, BDE atd.).

Vedle toho najdete na doprovodném CD i velkorysý dárek firmy Inprise — úplnou instalaci Borland C++ Builderu 4.0 Enterprise se 60denním omezením.

Typografické konvence

Osobně se domnívám, že každý průměrně inteligentní čtenář pochopí tyto konvence bez jakéhokoli výkladu po letmém pohledu na kteroukoli stránku knihy. Redakce počítačové literatury však striktně požaduje zařazení odstavce o typografických konvencích, a proto jej najdete i zde. V této knize dodržuji následující konvence:

`while`	Neproporcionálním písmem píši ukázky programů, klíčová slova, identifikátory a jména souborů.
About	Bezpatkové písmo používám pro názvy dialogových oken, tlačítek, příkazy nabídek atd. Vzhledem ke způsobu programování v C++ Builderu ovšem nelze položky této kategorie vždy dobře odlišit od položek předchozí skupiny; nepředpokládám ale, že by to mohlo způsobit jakékoli problémy.
Důležité	Kurziva znamená vždy upozornění na něco důležitého.
CTRL+F9	Kapitálky označují klávesové zkratky.

Terminologie

Pokud je to možné, používám českou terminologii. To znamená, že hovořím například o vláknech, nikoli o threadech. Původní termín ovšem uvádím (alespoň při prvním výskytu) v závorkách. Jsem přesvědčen, že česká terminologie může usnadnit pochopení, oč jde.

Miroslav Virius
katedra matematiky FJFI ČVUT
virius@km1.fjfi.cvut.cz

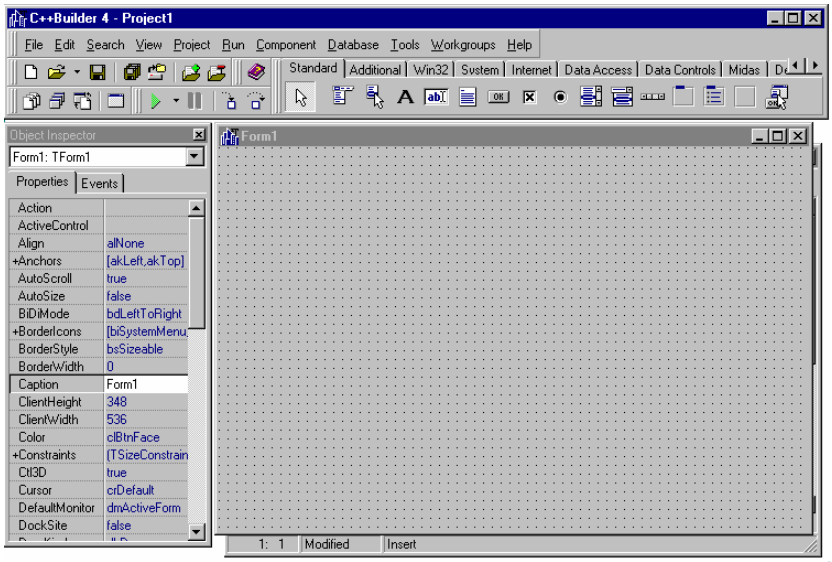
1. První rozhlédnutí

Stalo se zvykem, že knihy o programování, a zejména knihy, které se nějak zabývají jazyky C a C++, začínají programem, který vypíše nějaký vtipný text — nejlépe „Hello, world“ — a skončí. My si dovolíme tuto tradici porušit, ale ne příliš: k programu tohoto druhu se dostaneme v následující kapitole. Před tím se trochu porozhlédneme po programovém prostředí C++ Builderu a naučíme se základní operace s ním. Přitom vytvoříme program, který na obrazovce otevře prázdné okno — nic víc.

1.1 Prázdné okno

Borland C++ spouštíme — podobně jako jiné programy pod Windows — z nabídky Start | Programy | Borland C++ Builder 4 | C++ Builder 4. Ovšem protože s ním budeme pracovat pravidelně, je rozumné zřídit si pro něj na pracovní ploše zástupce s spouštět jej dvojklikem na tuto ikonu.

Po spuštění C++ Builderu by se na vašem monitoru mělo objevit něco podobného jako na obrázku 1.1.



Obr. 1.1 Prostředí C++ Builderu po spuštění

Pokud uvidíte něco jiného, použijte příkaz File | New Application z hlavní nabídky.

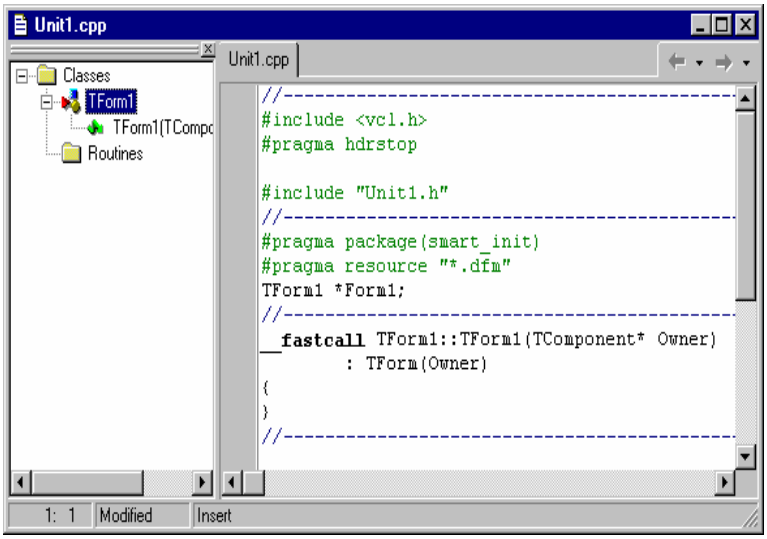
Prostředí se skládá ze tří oken (vlastně ze čtyř, ale čtvrté je zatím skryto). V horní části obrazovky je hlavní okno Builderu. Kromě nabídky a panelu nástrojů je na něm tzv. *paleta*, které obsahují stránky[1] s *vizuálními komponentami* — připravenými součástmi, které budeme v programech používat. Při používání nástrojů na panelu a komponent na paletě nabízí prostředí bublinovou nápovědu: Zastavíme-li se kurzorem myši nad některým tlačítkem nebo komponentou, objeví se „bublina“ se stručnou informací oč jde.

Pod hlavním oknem vpravo je nápadné prázdné šedé okno s titulkem Form1; to jo budoucí hlavní okno naší aplikace, které slouží jako základ vizuálního návrhu programu. (V terminologii Builderu se okno nazývá formulář, anglicky form.) Poznamenejme, že v tuto chvíli již prostředí vytvořilo program, který po spuštění otevře na obrazovce právě toto okno.

Posledním viditelným oknem je inspektor objektů (Object Inspector); vidíte ho vlevo pod hlavním oknem Builderu. Inspektor ukazuje některé vlastnosti komponent a umožňuje měnit je. V této chvíli ukazuje vlastnosti hlavního okna.

Na obrazovce je ovšem ještě čtvrté (a vlastně i páté) okno; skrývají se pod hlavním oknem budoucí aplikace a vidíte je na obr. 1.2. Vpravo je okno textového editoru. Obsahuje část zdrojového programu, který prostředí vytvořilo na základě vizuálního návrhu (zatím prázdného).

[1] Místo „stránka palety Standard“ budeme často říkat jen „paleta Standard“ apod.

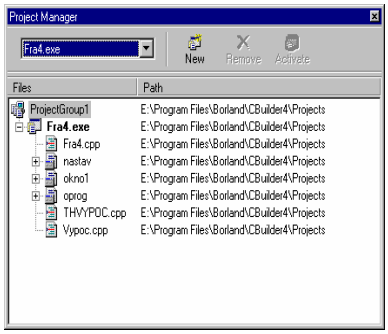


Obr. 1.2 Okno textového editoru

V levé části je okno nástroje, který se nazývá *průzkumník tříd* (Class Explorer). Pomocí stromu znázorňuje třídy, jejich složky a další součástí našeho programu. Dvojklik na jméno třídy nebo její složky v průzkumníkovi tříd způsobí, že se v editoru otevře odpovídající část zdrojového programu. Vedle toho, že usnadňuje orientaci v rozsáhlých projektech, umožňuje i přidávat do existujících tříd nové složky. Podrobněji si o průzkumníkovi tříd povíme v kap. 12.

Projekt

Podobně jako v jiných vývojových nástrojích řídíme i v C++ Builderu překlad a sestavování aplikací pomocí tzv. projektů. Builder po spuštění (nebo poté, co zadáme příkaz File | New) vytvoří nový projekt s implicitním názvem ProjectN, kde N je nějaké číslo. Tento projekt zahrnuje funkční kostru budoucí aplikace, kterou prostředí Builderu vygenerovalo. Podíváme se nejprve na jeho strukturu: Příkazem View | Project Manager (nebo klávesovou zkratkou CTRL + ALT + F11) vyvoláme okno správce projektů.



Obr. 1.3 Správce projektů

Po rozvinutí stromu v tomto okně dostaneme strukturu, kterou vidíme na obr. 1.3. Náš projekt má zatím implicitní jméno Project1 a obsahuje soubory Project1.cpp, Unit1.cpp a Form1.dfm. Project1.cpp obsahuje hlavní program, Unit1.cpp obsahuje část programu, která se stará o hlavní okno aplikace. Form1.dfm je datový soubor obsahující vlastnosti hlavního okna (je to vlastně druh prostředků, anglicky označovaných resources). Vedle zdrojového souboru Unit1.cpp vytvořilo prostředí ještě hlavičkový soubor Unit1.h, který se ale v okně projektu nezobrazuje.

Dříve, než budeme pokračovat v práci, uložíme vytvořený projekt a při tom pojmenujeme jednotlivé soubory. Z hlavní nabídky vybereme příkaz File | Save All nebo použijeme tlačítko Save All na panelu nástrojů. Prostředí se nás zeptá, kam a pod jakým jménem chceme uložit[2] soubor Unit1.cpp;

[2] Pozor: Bezprostředně po instalaci nabízí prostředí z zpravidla uložení souborů do adresáře Bin s programovými soubory C++ Builderu, což je nejspíš nejhorší myslitelná volba. Je rozumné použít např. adresář ..CBuilder4\Projects, který se automaticky vytvoří při instalaci a který je k tomuto účelu určen.

nazveme ho třeba okno.cpp a uložíme ho do adresáře Projects\01. Přitom se zároveň uloží i hlavičkový soubor (okno.h) a soubor s popisem okna (okno.dfm). Poté se nás prostředí zeptá na jméno projektu; použijeme např. jméno Prvni.

Nyní můžeme vytvořený program přeložit a spustit. Překlad spustíme příkazem Project | Make první nebo pomocí klávesové zkratky CTRL+F9. Protože jsme sami zatím nenapsali ani písmeno zdrojového textu, lze očekávat, že překlad proběhne bez chyb. Poté program spustíme; k tomu použijeme příkaz Run | Run, klávesovou zkratku F9 nebo tlačítko Run z panelu nástrojů (je na něm zelený trojúhelník ▶). Objeví se prázdné okno, které má všechny běžné vlastnosti oken v aplikacích pro Windows: Lze je přemísťovat, měnit jeho velikost, minimalizovat, maximalizovat. Stiskem tlačítka ✕ v pravém horním rohu (nebo třeba dvojklikem na ikonu v levém horním rohu) tento program ukončíme.

Soubory

Podívejme se nyní do adresáře ...CBuilder4\Projects, do něhož jsme uložili náš projekt. Najdeme v něm následující soubory:

okno.cpp	Zdrojový soubor pro okno programu
okno.h	Hlavičkový soubor k souboru okno.cpp
okno.dfm	Binární soubor s popisem vlastností okna
okno.obj	Přeložený zdrojový soubor okno.cpp
prvni.cpp	Zdrojový soubor s hlavním programem
prvni.bpr	Soubor s popisem projektu
prvni.obj	Přeložený zdrojový soubor prvni.cpp
prvni.tds	Soubor s informacemi potřebnými pro ladění
prvni.res	Soubor s přeloženými prostředky (resources)
prvni.exe	Výsledný spustitelný soubor, tedy hotový program

Dále tu najdeme soubory s příponou začínající znakem „~“; to jsou záložní kopie, které prostředí vytvořilo při ukládání.

Soubor .cpp, spolu s odpovídajícím souborem .h a případně *.dfm tvoří logický celek, který označujeme jako modul (anglicky unit).

Poznamenejme, že při přenosu projektu na jiný počítač musíme vzít soubory *.cpp, *.h, *.dfm, *.res a soubor *.bpr. U dalších projektů mohou přibýt ještě soubory *.rc, *.def a další.

1.2 Program

Nyní prozkoumáme program, který nám C++ Builder vytvořil.

Okno

Podívejme se nejprve na soubor okno.cpp. Dvojklikem na položku okno.cpp v okně správce projektu přejdeme do editoru, ve kterém uvidíme následující zdrojový text:

```
#include <vcl.h>
#pragma hdrstop

#include "okno.h"
#pragma package(smart_init)
#pragma resource "*.dfm"

TForm1 *Form1;
__fastcall TForm1::TForm1(TComponent* Owner)
  : TForm(Owner)
{}
```

Výpis souboru okno.cpp

Budeme-li najednou pracovat na více projektech, uložíme pochopitelně nejspíš každý z nich do samostatného podadresáře. V kap. 12 si ukážeme, jak změnit implicitní adresář.

První řádek obsahuje direktivu #include, kterou se do programu vkládá hlavičkový soubor obsahující deklarace tříd vizuálních komponent — tedy předdefinovaných tříd popisujících okno aplikace a komponenty na paletě. Poznamenejme, že soubor vcl.h vlastně jen řídí vkládání velkého množství dalších hlavičkových souborů. Direktiva #pragma hdrstop ve druhém řádku ukončí vytváření souboru s předkompilovanými záhlavími.

Následující direktiva #include vkládá hlavičkový soubor obsahující definici třídy TForm1 (tj. třídy našeho hlavního okna). Pak následujíc dvě direktivy #pragma. První z nich ovlivňuje inicializaci knihoven, druhá říká, že k němu patří také soubor se stejným jménem a s příponou .dfm, který popisuje okno.

Následuje deklarace ukazatele na instanci třídy TForm1, která v programu reprezentuje hlavní okno aplikace. Zbytek není nic jiného než deklarace konstruktoru třídy TForm1. Jediná věc, kterou tento konstruktor udělá, je, že zavolá konstruktor předka (třídy TForm) a předá mu svůj parametr.

Dále se podíváme na hlavičkový soubor okno.h. Nejjednodušší způsob, jak si ho otevřít, je kliknout pravým tlačítkem myši v okně editoru, ve kterém jsme si otevřeli soubor okno.cpp, a v příruční nabídce zvolit možnost Open Source/Header File nebo použít klávesovou zkratku CTRL+F6. (Mohli bychom také zadat v hlavní nabídce příkaz File | Open ..., v rozbalovacím seznamu Files of type předepsat možnost C files a pak zvolit okno.h.) Uvidíme následující program:

```
#ifndef oknoH
 #define oknoH

#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>

class TForm1 : public TForm
{
__published:                    // IDE-managed Components
 private:                       // User declarations
 public:                        // User declarations
          __fastcall TForm1(TComponent* Owner);
};
extern PACKAGE TForm1 *Form1;
#endif
```

Výpis souboru okno.h

V úvodu najdeme direktivy #ifndef a #define, které zabezpečují, že tento soubor bude překládán jen jednou. Následující direktivy #include vkládají hlavičkové soubory s definicemi knihovních tříd.

Pak následuje definice třídy[3] hlavního okna naší aplikace, TForm1; ta je odvozena od předdefinované třídy TForm. Jediná složka, která je zde nově definována, je veřejně přístupný konstruktor.

Poznámky

- ✧ Klíčové slovo __fastcall v deklaraci konstruktoru je borlandské rozšíření jazyka C++, které specifikuje tzv. registrovou volací konvenci. Zde mj. předepisuje, že se některé z parametrů konstruktoru budou předávat v registrech procesoru. Tato volací konvence je povinná pro metody tříd z knihovny vizuálních komponent.
- ✧ Také klíčové slovo __published je borlandské rozšíření jazyka C++. Jeho přesný význam si vysvětlíme později, zatím postačí vědět, že složky deklarované v této sekci jsou veřejně přístupné (jako složky v sekci public) a že do této sekce bude C++ Builder zapisovat deklarace komponent, které do okna vložíme. Sami toto klíčové slovo zatím nebudeme používat; najde uplatnění vlastně až při psaní nových vizuálních komponent.
- ✧ Komentář za klíčovým slovem __published nás upozorňuje, že tuto část deklarace třídy spravuje C++ Builder. Sem bychom neměli zasahovat, tj. přidávat sem deklarace nebo je odsud odstraňovat. Mohli bychom tím narušit funkci Builderu; v krajním případě by se mohlo stát, že prostředí odmítne s naším programem pracovat.

[3] Čtenáře, kteří znají základy programování ve Windows API, bych rád upozornil na možnost nedorozumění. „Třída okna“ bude v této knize znamenat objektový typ, který popisuje některé z oken aplikace, nikoli třídu ve smyslu API.

Popis okna (soubor DFM)

V souboru okno.dfm jsou v binární podobě uloženy vlastnosti okna (jeho velikost, barva, poloha na obrazovce, použité písmo atd.). Pokud chceme, můžeme si je zobrazit v textové podobě. Je-li toto okno v prostředí Builderu aktivní, stačí vyvolat stisknutím pravého tlačítka myši příruční nabídku a z ní si vybrat položku View as Text. Můžeme také použít klávesovou zkratku ALT+F12. Okno zmizí a místo něj se v editoru objeví následující popis:

```
object Form1: TForm1
 Left = 193
 Top = 108
 Width = 544
 Height = 375
 Caption = 'Form1'
 Font.Charset = DEFAULT_CHARSET
 Font.Color = clWindowText
 Font.Height = -11
 Font.Name = 'MS Sans Serif'
 Font.Style = []
 PixelsPerInch = 96
 TextHeight = 13
end
```

Textový popis okna v souboru okno.dfm

Tento popis sice lze editovat, ale vhodnější je obvykle měnit tyto vlastnosti v inspektoru objektů. Navíc neuvážené zásahy zde mohou způsobit, že prostředí bude hlásit podivné chyby a případně že s projektem odmítne pracovat. (Podobně se chovají i jiné vizuální nástroje, které automaticky generují zdrojový kód.)

K původnímu zobrazení se vrátíme příkazem View as Form z příruční nabídky nebo klávesovou zkratkou ALT+F12.

Hlavní program

Zbývá podívat se na hlavní program. C++ Builder ho sám od sebe nezobrazil, takže si pomůžeme dvojklikem na položku prvni.cpp v okně správce projektu a uvidíme následující zdrojový text:

```
#include <vcl.h>
#pragma hdrstop

USERES("prvni.res");
USEFORM("okno.cpp", Form1);

WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
  try
  {
     Application->Initialize();
     Application->CreateForm(__classid(TForm1), &Form1);
     Application->Run();
  }
  catch (Exception &exception)
  {
     Application->ShowException(&exception);
  }
  return 0;
}
```

Výpis souboru prvni.cpp

Za úvodními direktivami následují dvě makra, která zajistí připojení souboru s prostředky (prvni.res) a umožní vytvoření hlavního okna.

Jak jistě víte, programy pro Windows nepoužívají funkci main(), ale funkci WinMain(). Zde ji ovšem najdeme v poněkud netradičním tvaru. Vlastní aplikace je v Borland C++ Builderu totiž tvořena instancí Application třídy TApplication; tato instance je připravena v knihovně, takže hlavní program pouze volá její metody Initialize(), CreateForm() a Run(). První z nich se postará o nezbytné inicializace, druhá vytvoří hlavní okno aplikace a třetí zabezpečí vlastní běh programu. To znamená, že obsahuje mj. cyklus zpráv, který zajistí předávání zpráv od Windows našemu programu.

Všechna tato tři volání jsou uzavřena do bloku try, takže pokud se některá z operací nepodaří a skončí výjimkou, přejde program do bloku catch, vypíše zprávu o chybě a skončí. Jinak program skončí, jakmile skončí metoda Run() — tj. jakmile se uzavře hlavní okno našeho programu.

Poznámky

- ✧ Hlavní program v této podobě je dostačující pro naprostou většinu případů. To znamená, že ho nejspíš nebudeme potřebovat měnit.
- ✧ Makro USEFORM se rozvine v předběžnou deklaraci třídy TForm1 a deklaraci ukazatele Form1 typu TForm1*.
- ✧ Operátor __classid je opět borlandské rozšíření jazyka C++. Jeho význam si vysvětlíme v kapitole 11; při běžném programování jej však nebudeme potřebovat, a proto na něj můžete zatím s klidem zapomenout.

Projektový soubor

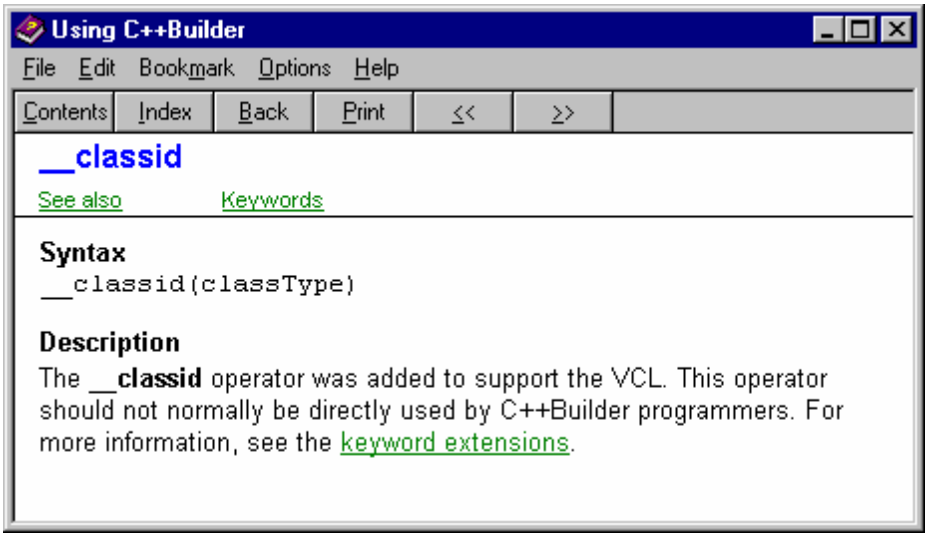
Na závěr se zastavíme u souboru prvni.bpr, který obsahuje popis našeho projektu. To je textový soubor, jehož první část je vlastně makefile, tedy soubor, který se používá pro řízení překladu u překladačů spouštěných z příkazové řádky. Druhá část připomíná inicializační soubory (.ini) aplikací pro Windows a jsou v ní uložena nastavení prostředí Builderu. Tento soubor spravuje prostředí Builderu, přímo do něj (téměř nikdy) přímo nezasahujeme.

1.3 Nápověda

Nápověda k Borland C++ Builderu nabízí v podstatě stejné možnosti jako nápověda v jiných programech pro Windows. To znamená, že v ní můžeme vyhledávat informace podle obsahu dokumentace, podle rejstříku nebo tak, že zadáme vyhledání určitého slova. (K tomu nám poslouží příkazy z hlavní nabídky Help | Contents resp. Help | Index.)

Vedle toho můžeme použít kontextovou nápovědu.

- ✧ Jestliže se objeví libovolné dialogové okno, dostaneme po stisknutí klávesy F1 informace o možnostech, které v něm máme.
- ✧ Jestliže vybereme myší komponentu na některé ze stránek palety a stiskneme klávesu F1, dostaneme informace o této komponentě (tj. o její třídě). Klikneme-li v inspektoru objektů na některou z vlastností nebo událostí komponenty, dostaneme po stisknutí F1 informace o této vlastnosti.
- ✧ Je-li textový kurzor v editoru na některém klíčovém slovu nebo na identifikátoru některé knihovní funkce nebo třídy, získáme stisknutím klávesy F1 informace o této součásti jazyka C++ v Builderu (obr. 1.4). Pokud najde Builder více podobných témat, zeptá se, které nás vlastně zajímá.
- ✧ Vybereme-li myší některé z chybových hlášení nebo z varování, vyvoláme pak stisknutím klávesy F1 nápovědu k této chybě.
- ✧ Okno s nápovědou uzavřeme např. stisknutím klávesy ESC.



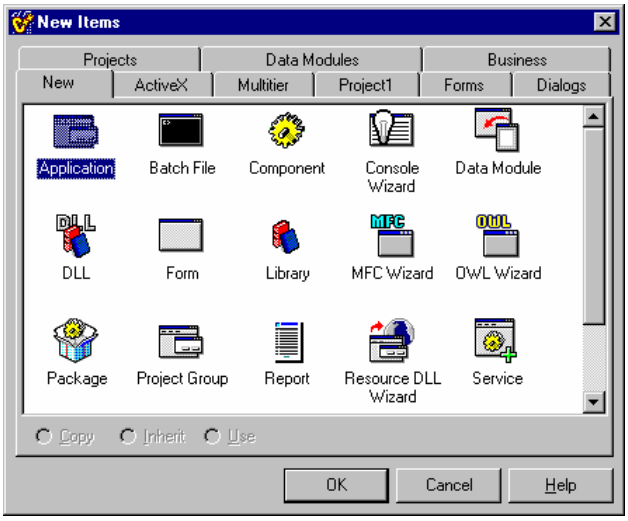
Obr. 1.4 Kontextově citlivá nápověda

2. Setkání s komponentami (volební program)

V minulé kapitole jsme se trochu seznámili s prostředím a získali jsme představu, jak s ním zacházet; to znamená, že už je nejvyšší čas pustit se do programování. Začneme programem, který okně zobrazí vhodný nápis, nejlépe předvolební heslo, a pak budeme tento program vylepšovat. Přitom se především naučíme pracovat s komponentami.

2.1 Okno s nápisem

Vytvoříme nový projekt příkazem nabídky File | New. Objeví se okno zásobníku objektů (anglicky object repository, obr. 2.1); přejdeme na kartu New a zvolíme položku Application. Nyní jsme ve stejné situaci jako v minulé kapitole, takže uložíme vytvořený projekt (např. stisknutím tlačítka Save All). Projekt pojmenujeme hello1, modul pro okno třeba opět okno.cpp.



Obr. 2.1 Zásobník objektů

Nejprve upravíme vlastnosti hlavního okna. První, co změníme, je titulek — chceme, aby tam místo Form1 byl nápis „Bobbyho přesvědčení“, neboť do okna hodláme vložit známý citát z knihy Murphyho zákon. K tomu použijeme inspektora objektů (obr. 2.2).

V horní části inspektora je rozbalovací seznam objektů v aktivním okně. Tento seznam vždy ukazuje aktivní objekt, tj. objekt, jehož vlastnosti inspektor objektů právě zobrazuje. Zatím je v zde jediná položka, okno, které se jmenuje[4] Form1 a je typu TForm1.

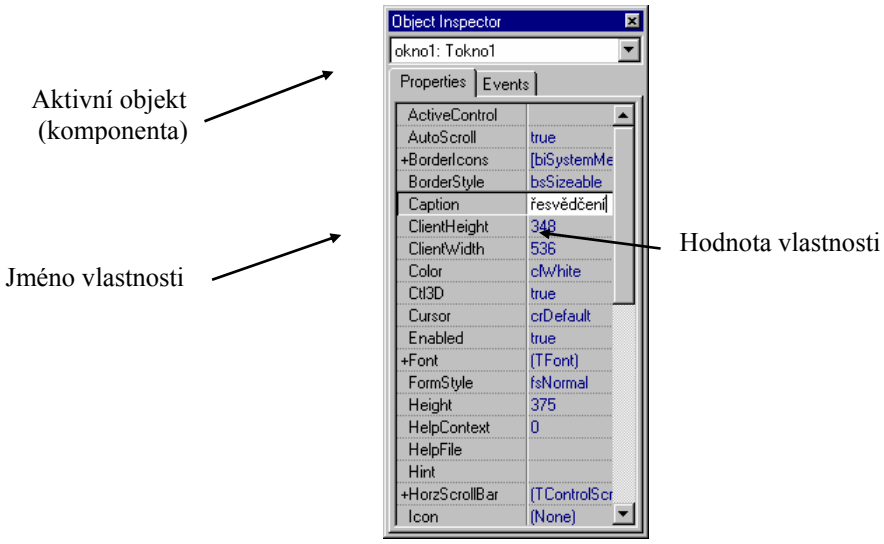
Pod seznamem objektů jsou dvě karty, Properties (vlastnosti) a Events (události). Kliknutím na odpovídající záložku přejdeme na kartu vlastností (pokud už není na vrchu). Tato karta má dva sloupce. V levém jsou jména vlastností, v pravém pak jejich hodnoty. Vyhledáme vlastnost Caption, která určuje titulek okna, a do pravé části zapíšeme požadovaný nápis. Všimněte si, že zároveň s tím, jak píšeme, se mění titulek okna ve vizuálním návrhu naší aplikace.

Dále změníme identifikátor, pod kterým se bude objekt představující toto okno objevovat v programu. Vyhledáme vlastnost Name a změníme ji z Form1 na okno1[5]. Tím změníme jméno ukazatele, jehož prostřednictvím budeme v programu zacházet s hlavním oknem, a také jméno typu (třída okna se nyní bude jmenovat Tokno1). Tato změna se ihned odrazí jak v inspektoru objektů tak i ve vygenerovaném zdrojovém textu.[6]

[4] Přesněji, Form1 se jmenuje ukazatel na instanci třídy TForm1, která představuje hlavní okno našeho programu.

[5] Bylo by logické pojmenovat naše okno okno. To ale C++ Builder nedovoluje — okno se z jakýchsi záhadných důvodů nesmí jmenovat stejně jako soubor, ve kterém je uloženo. (Přesněji, jde o důsledek kompatibility s Delphi.)

[6] V dalším textu budeme často místo „jménu ukazatele na okno nebo komponentu“ říkat jen „jméno okna nebo komponenty“. Vzhledem k tomu, že s komponentami smíme pracovat pouze prostřednictvím ukazatelů, nehrozí nedorozumění.

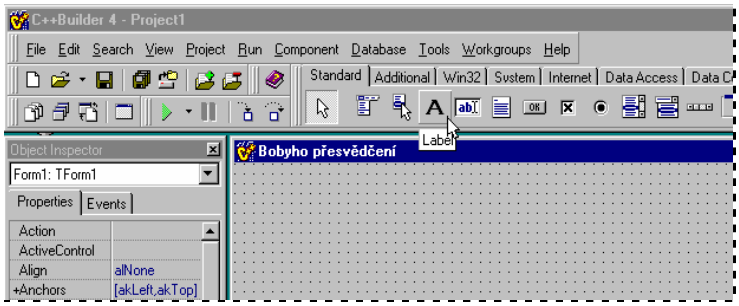


Obr. 2.2 Inspektor objektů: měníme vlastnosti formuláře

Naším dalším krokem bude vytvoření nápisu, který má náš program zobrazovat; k tomu použijeme komponentu Label, kterou najdeme na paletě na stránce Standard. (Pokud tato stránka není aktivní, přejdeme na ni kliknutím na záložku s nápisem Standard.)

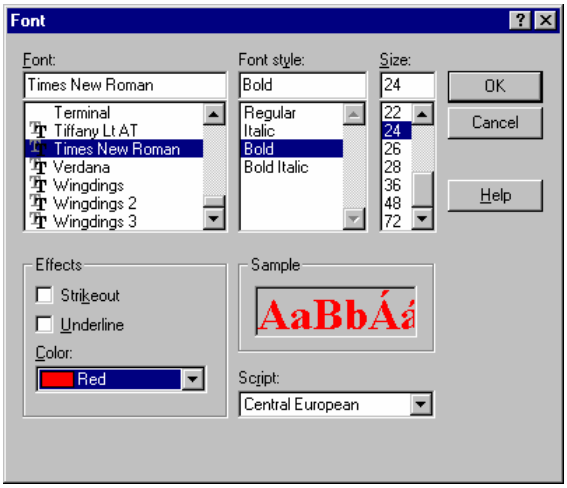
Nejprve klikneme na ikonu komponenty Label; tím ji vybereme. Pak klikneme přibližně doprostřed formuláře (okna) našeho budoucího programu a tím vložíme odpovídající komponentu svého programu. C++ Builder na toto místo vloží nápis Label obklopený „úchyty“ — černými body, které umožňují komponentu myší přemísťovat a měnit její velikost. V seznamu objektů v inspektoru objektů se objeví položka Label1: TLabel, která nám říká, že v programu budeme s touto komponentou typu TLabel zacházet pomocí ukazatele s identifikátorem Label1 (pokud ho nepřejmenujeme).

Nyní nastavíme vlastnosti této komponenty. Nejprve změníme jméno ukazatele na ni tím, že vlastnosti Name přidělíme hodnotu heslo. Pak změníme nápis, který tato komponenta zobrazuje. Vyhledáme v inspektoru objektů vlastnost Caption a do okénka s hodnotou napíšeme větu „Zmatek vládne i bez ministrů.“ I tentokrát se nápis v okně ihned změní.



Obr. 2.3 Vybereme komponentu Label a umístíme ji do okna aplikace

Dále bychom si přáli, aby se naše volební heslo zobrazovalo vždy uprostřed okna. Toho dosáhneme v několika krocích.

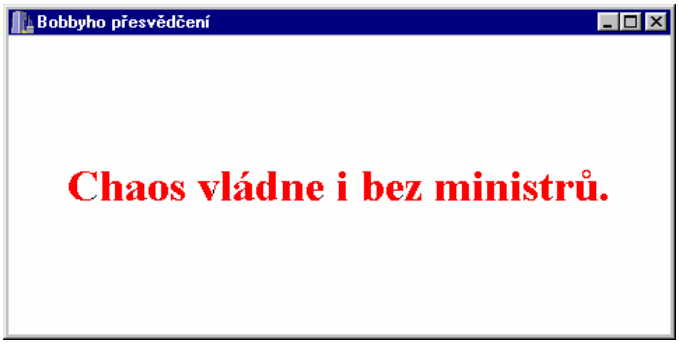


Obr. 2.4 Určujeme písmo

Nejprve změníme hodnotu vlastnosti Align. Po kliknutí na tuto položku se vedle implicitní hodnoty alNone objeví tlačítko se šipkou dolů (▽), které naznačuje, že si budeme moci vybrat z několika předdefinovaných hodnot; jejich seznam se objeví po kliknutí na toto tlačítko. Zvolíme hodnotu alClient, která způsobí, že komponenta vyplní celou klientskou oblast okna. Pak nastavíme vlastnost Alignment na taCenter; nápis bude nyní umístěn podélně ve středu okna. Nakonec nastavíme vlastnost Layout na tlCenter a tím nápis vycentrujeme i ve svislém směru.

Nápis je nyní sice ve středu okna, je ale malý a i jinak nevýrazný — funkci předvolebního sloganu by určitě nesplnil. Proto změníme typ písma, jeho velikost a barvu. Najdeme si vlastnost Font a kliknutím do této kolonky inspektora objektů ji aktivujeme. Vpravo od implicitní hodnoty se objeví tlačítko se třemi tečkami; jeho stisknutím vyvoláme dialogové okno Font, v němž předepíšeme např. písmo Times New Roman, tučné (bold), velikost 24 bodů a červenou barvu (obr. 2.4).

Nakonec změníme barvu pozadí nápisu na bílou, abychom heslo ještě více zdůraznili. K tomu nám poslouží vlastnost Color; implicitní hodnotu clBtnFace změníme na clWhite. (Předdefinované hodnoty barev mají názvy složené z předpony cl a anglického jména barvy.) Tím jsme hotovi; projekt uložíme, přeložíme a spustíme. Výsledek ukazuje obr. 2.4. Můžete se přesvědčit, že nápis zůstává uprostřed okna, i když změníme jeho velikost, a že text v něm se neztratí, i když okno našeho programu na čas překryjeme jiným. (Úplný projekt najdete na CD adresáři KAP02\01.)



Obr. 2.5 Výsledek naší práce

Program

Ani náš druhý program zatím nevyžadoval „skutečné“ programování; ale nebojte se, dojde na ně také, a už brzy. Nicméně nyní se podíváme, jak vypadá program, který pro nás C++ Builder vytvořil.

Hlavní program obsahující funkci WinMain(), uložený v souboru hello1.cpp, je stejný jako v minulé kapitole, a proto jej zde nebudeme opakovat.

Podívejme se ale na soubor okno.h. Definice třídy Tokno1 má nyní tvar

```
class Tokno1 : public TForm
{
__published:
```

```
  TLabel *heslo;            // Ukazatel na přidanou komponentu
private:
public:                                 // Konstruktor
  __fastcall Tokno1(TComponent* Owner);
};
```

Definice třídy Tokno1 v souboru okno.h

Ve srovnání s předchozí kapitolou zde přibyla pouze jediná věc — deklarace ukazatele heslo typu Tlabel* v sekci __published. Také definice ukazatele na instanci této třídy (v souboru okno.cpp) se po přejmenování změnila:

```
Tokno1 *okno1;
```

Žádná z nastavených vlastností se však ani v jednom z těchto souborů neobjevila, ty najdeme až v souboru okno.dfm. Zde si ukážeme jen část výpisu:

```
object okno1: Tokno1
 ...
 Caption = 'Bobbyho přesvědčení'
 ...
 object heslo: TLabel
  ...
  Align = alClient
  ...
  Font.Name = 'Times New Roman'
  Font.Style = [fsBold]
  ...
 end
end
```

Zkrácený výpis textového popisu okna (souboru okno.dfm)

Všimněte si, že v programu se s komponentou vloženou do okna pracuje — podobně jako s celým oknem a se všemi komponentami — prostřednictvím ukazatele, který je atributem (datovou složkou) třídy okna. V souboru okno.dfm je popis této komponenty vnořen do popisu okna.

2.2 Vylepšujeme svůj program

Pokusme se nyní svůj program trochu vylepšit. Abychom si ale nějakým nešikovným zásahem nepokazili hotovou práci, budeme pracovat s kopií původního projektu. Mohlo by se zdát logické, že k tomu použijeme příkaz File | Save Project As... a zadáme jméno hello2. Ale pozor: Tento příkaz uloží na nové místo a pod novým jménem pouze soubor hello2.cpp s hlavním programem, soubor hello2.bpr s popisem projektu a soubor hello2.res. *Přitom soubor s popisem projektu upraví tak, že bude odkazovat na moduly pro jednotlivá okna z původního projektu*.

To se nám nehodí, a proto použijeme jiný postup: Nejprve uzavřeme příkazem File | Close All aktuální projekt. Pak zkopírujeme soubory *.cpp, *.bpr a *.dfmdo jiného adresáře a příkazem File Open Project... otevřeme nový projekt. Projektový soubor *.bpr obsahuje pouze relativní cesty, takže nenastanou problémy.

Zalamování textu do okna

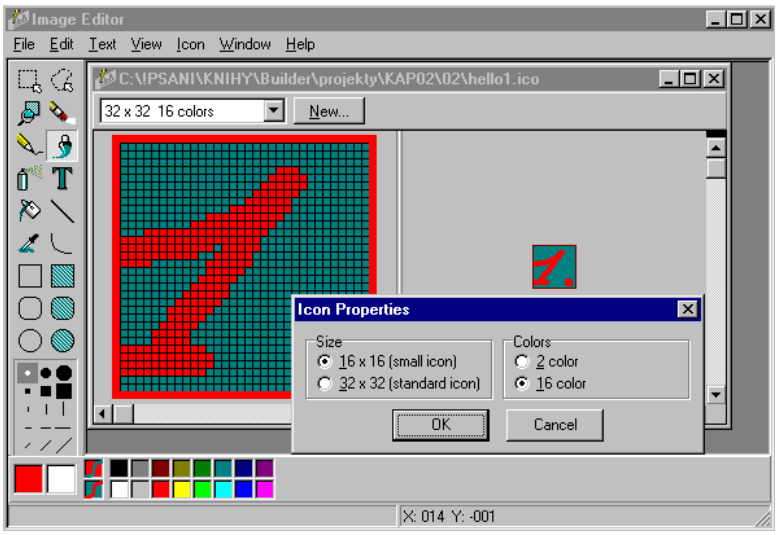
První nedostatek, který by nám mohl vadit, je, že pokud okno příliš zmenšíme, nebude nápis vidět celý. Tomu lze do značné míry předejít, jestliže přikážeme, aby se text v případě potřeby zalamoval do více řádků; nastavíme tedy vlastnost WordWrap na true.

Ikona programu

Také standardní ikona, kterou nám C++ Builder k programu připojil, se nám příliš nelíbí; nahradíme ji tedy nějakou lepší. Nejprve si ji nakreslíme. K tomu použijeme editor obrázků[7] (Image Editor), který si spustíme z nabídky Builderu Tools | Image Editor. V nabídce editoru obrázků si příkazem File | New

[7] Tento nástroj slouží k vytváření ikon, kurzorů a bitových map. Ostatní prostředky, jako jsou menu, editační pole nebo dialogová okna, vytváříme pomocí komponent; tabulku řetězců (string table) vytváříme pomocí klíčového slova resourcestring.

Icon File... předepíšeme, že chceme vytvořit novou ikonu a uložit ji v samostatném souboru .ico. Objeví se dialogové okno Icon Properties (vlastnosti ikony), v němž určíme, že chceme standardní ikonu (32 × 32 bodů) se 16 barvami. Po stisknutí tlačítka OK můžeme kreslit. Zacházení s tímto editorem je podobné jako zacházení s jinými jednoduchými bitmapovými editory pod Windows (Paintbrush atd.) V levé části okna kreslíme ve zvětšeném měřítku, v pravé části okna vidíme své dílo ve skutečné velikosti (obr. 2.6). Zde jsme jako ikonu zvolili číslici 1, neboť jde o náš první skutečný program. (Nebo kvůli volbám?)



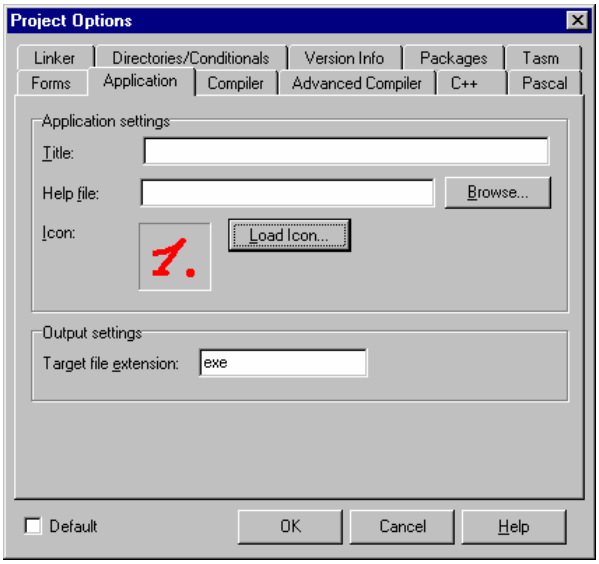
Obr. 2.6 malujeme ikonu v editoru obrázků

Po dokončení ikonu uložíme příkazem File | Save... pod názvem hello1.ico do stejného adresáře jako ostatní soubory projektu.

Tím ale nejsme s kreslením hotovi. U aplikací v prostředí Win32 je obvyklé, že mají dvě ikony — „velkou“ (32 × 32 bodů) a „malou“ (16 × 16 bodů). Stiskneme proto tlačítko New v horní části okna s ikonou a v dialogovém okně Icon Properties si poručíme ikonu o rozměru 16 × 16 bodů se 16 barvami.

Namalujeme také malou ikonu, soubor uložíme a editor obrázků uzavřeme.

Dalším krokem je připojení ikony k programu. I to je velice jednoduché: Příkazem Project | Options... v prostředí Builderu vyvoláme dialogové okno Project Options (obr. 2.6), v něm přejdeme na kartu Application a stiskneme tlačítko Load Icon. Builder začne s hledáním souboru s ikonou v domovském adresáři projektu, takže soubor hello1.ico najdeme bez problémů. Nenechte se zmást skutečností, že se změna ikony neprojeví hned po uzavření okna. *Změna ikony se projeví až při spuštění aplikace; ikona bude uložena v souboru* .res.



Obr. 2.7 Definujeme ikonu aplikace

Tlačítko pro ukončení programu

Nyní se konečně dostaneme k troše programování. Přidáme do našeho programu tlačítko, po jehož stisknutí program skončí. Na paletě Additional si vybereme komponentu BitBtn (tlačítko s bitovou mapou) a umístíme ho do pravého dolního rohu okna naší aplikace.

Pak určíme jeho vlastnosti. Nazveme ho konec. Tím, že nastavíme vlastnost Kind na hodnotu bkClose, umístíme na tlačítko standardní ikonu používanou na tlačítku Cancel. Pak do pole pro hodnotu vlastnosti Caption napíšeme slovo „&Konec“ a tím určíme nápis na tlačítku[8]. Nastavíme-li ještě vlastnost Layout na hodnotu blGlyphLeft, stanovíme tím, že ikona (glyph) bude na tlačítku nalevo od textu[9]. Nakonec nastavíme na true ještě vlastnosti Default a Cancel a tím zajistíme, že naše tlačítko bude reagovat nejen na stisknutí myší, ale i na stisknutí kláves ENTER a ESC.

Jestliže nyní program přeložíme a spustíme, bude na něm sice nové tlačítko s nápisem a s ikonou, ale nebude fungovat — po stisknutí se nic nestane. Musíme ještě naprogramovat jeho odezvu na stisknutí.

Přejdeme v inspektoru objektů na kartu Events (události). Hned v prvním řádku najdeme událost OnClick; dvojklikem do tohoto pole (nebo dvojklikem na tlačítko ve vizuálním návrhu aplikace) přejdeme do editoru zdrojového programu, ve kterém nám C++ Builder připravil kostru funkce Tokno1::konecClick(). Tuto funkci bude program volat, jestliže tlačítko stiskneme. (Budeme o ní hovořit jako o *funkci pro obsluhu události „stisknutí tlačítka“* neboli o *handleru ošetřujícím tuto událost*.)

Protože chceme, aby náš program po stisknutí tlačítka skončil, zavoláme v handleru metodu Close(). Tato metoda uzavře okno, a protože jde o hlavní okno programu (ostatně jiné náš program zatím nemá), způsobí také, že program skončí. Celý handler bude vypadat takto:

```
void __fastcall Tokno1::konecClick(TObject *Sender)
{
  Close();
}
```

Poznamenejme, že Close() je metoda, kterou třída Tokno1 zdědila od svého předka, třídy TForm.

Nyní můžeme program přeložit a spustit.

Chyby při překladu

Pokud v programu uděláme nějakou chybu, skončí překladač po dokončení překladu modulu. Chyby a varování se vypisují ve spodní části okna editoru.

Jedním kliknutím na hlášení o chybě v tomto okně se tato zpráva vybere; dalším kliknutím se v editoru zobrazí místo, kde překladač daný problém zjistil (to se ale nemusí přesně shodovat s místem, kde chyba opravdu je).

Jestliže vybereme zprávu o chybě nebo varování a stiskneme klávesu F1, nabídne nám prostředí k tomuto problému nápovědu.

Odstraňujeme komponentu a handler

Občas potřebujeme z programu některou z komponent odstranit. Zkusíme si to s čerstvě přidaným tlačítkem. (Doporučuji vzít opět kopii projektu, neboť v zápětí budeme pokračovat ve vylepšování našeho programu i s tlačítkem.)

Je to velice jednoduché: Ve vizuálním návrhu vybereme kliknutím myší komponentu, která se nám znelíbila, a stisknutím tlačítka DEL ji smažeme. O ostatní se postará prostředí samo. (Nenechte se svést pokušením odstranit „ručně“ ukazatel na tuto komponentu ve zdrojovém programu; to nestačí, neboť byste museli odstranit i její vlastnosti ze souboru .dfm.)

Tím, že smažeme komponentu, ale ještě neodstraníme handlery, které ošetřovaly některé z událostí vázaných na tuto komponentu. Jak uvidíme, jeden handler lze použít k obsluze několika

[8] Znak & před písmenem K říká, že ke „stisknutí“ tlačítka budeme noci použít i klávesovou zkratku ALT+K. Písmeno K se bude na tlačítku zobrazovat podtržené.

[9] Pokud se nám standardní ikona nelíbí, můžeme použít vlastní. Vlastnosti Kind ponecháme implicitní hodnotu bkCustom, klikneme v inspektoru objektů na řádek s vlastností Glyph a pak stiskneme tlačítko se třemi tečkami, které se objeví v pravé části pole s hodnotou. Poté se objeví dialogové okno, ve kterém určíme soubor s ikonou. Měla by to být bitová mapa (.bmp). Pokud nás výsledek neuspokojí, můžeme bitovou mapu z tlačítka odstranit tím, že v tomto dialogovém okně stiskneme tlačítko Clear.

různých událostí, dokonce i událostí generovaných různými komponentami, a proto ho Builder při vymazání komponenty neodstraní.

Chceme-li odstranit také handler konecClick(), smažeme jeho tělo (v našem případě příkaz Close();) a soubor uložíme; prostředí se postará o zbytek, tj. odstraní „kostru“ ze souboru .cpp a deklaraci handleru z odpovídajícího souboru .h. Prostředí totiž automaticky odstraňuje všechny „prázdné“ handlery.

Posuvníky a kotvy

Jestliže nyní okno zmenšíme tak, že tlačítko nebude vidět, přidá C++ Builder automaticky posuvníky (scroll bar), které nám umožní dostat se ke skrytému ovládacímu prvku. Pokud si přejeme, aby se posuvníky nezobrazovaly, nastavíme vlastnost AutoScroll formuláře na false. (Implicitně má tato vlastnost hodnotu true,. tj. posuvníky se zobrazují.)

C++ Builder 4 nabízí ale ještě jednu možnost, která v předchozích verzích nebyla — použít kotev komponent. Vybereme tlačítko a v inspektoru objektů najdeme vlastnost Anchors (kotvy). Kliknutím na křížek před jejím jménem ji rozvineme, tj. dostaneme se k dílčím vlastnostem, které ji tvoří. Tlačítko je implicitně ukotveno k levému a hornímu okraji — vlastnosti akLeft a akTop mají hodnotu true, takže se zachovává vzdálenost od těchto okrajů. Předepíšeme-li ukotvení k pravému a dolnímu okraji, tj. nastavíme-li akLeft a akBottom na true, zbývající na false, zůstane tlačítko při změnách velikosti okna viditelné.

Přeložený program s tlačítkem ukotveným k levému a hornímu okraji najdete v adresáři KAP02\02 pod názvem hello1.exe; program s tlačítkem ukotveným k pravému a dolnímu okraji najdete v témže adresáři pod názvem hello1a.exe.

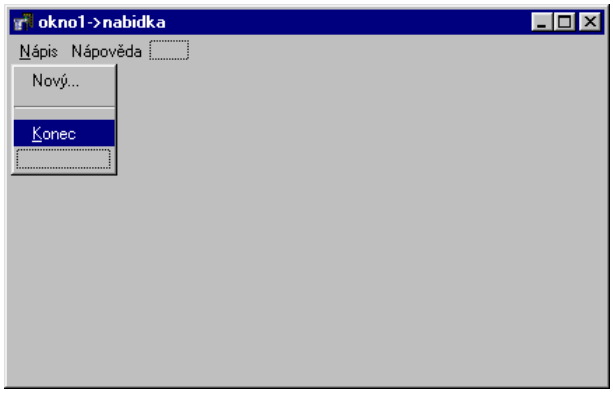
Nabídka neboli menu

Vraťme se k původnímu projektu před tím, než jsme smazali tlačítko Konec. Každý pořádný program pro Windows má nabídku (menu), a protože chceme svůj program využít k předvolební agitaci, je jasné, že musí mít menu také. Přidáme tedy nabídku s položkami Nápis | Nový..., Nápis | Konec a Nápověda | O programu... Jejich význam je jasný.

Na paletě Standard vybereme komponentu MainMenu, umístíme ji kamkoli do okna vizuálního návrhu programu a pojmenujeme ji nabidka (opět pomocí vlastnosti Name). Dvojklikem na ni pak vyvoláme editor nabídek (menu designer, obr. 2.8). To je okno, ve kterém vidíme budoucí nabídku a které zatím obsahuje jen jedinou hlavní položku. V inspektoru objektů tuto položku pojmenujeme napis a napíšeme na ni &Nápis (vlastnost Caption). Znak & opět znamená, že tuto položku půjde vybrat i klávesovou zkratkou ALT+N.

Nyní klikneme na položku Nápis v návrháři menu; tím se vytvoří podřízená položka (popup). Pojmenujeme ji Novy a napíšeme na ni Novy. Pak klikneme na prázdnou položku za položkou Nový a do její vlastnosti Caption napíšeme „-“ (minus). Tak vytvoříme separátor — vodorovnou čáru v menu, která výrazně odděluje různé skupiny položek. Pod separátorem ještě vytvoříme položku Konec.

Podobně kliknutím na prázdné políčko vedle položky Nápis vytvoříme novou položku v hlavním menu, kterou pojmenujeme napoveda a která ponese nápis Nápověda, a podřízenou položku Oprogramu1 s nápisem O programu.... Pak editor nabídek uzavřeme.



Obr. 2.8 Editor nabídek

Poznámky

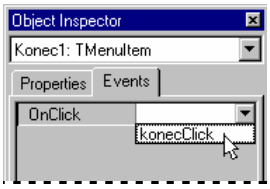
- ✧ Podívejme se na zdrojový text v souboru okno.h. V deklaraci třídy Tokno1 přibyl ukazatel na jednu instanci typu TMainMenu a šest ukazatelů na položku typu TMenuItem — pro každou položku nabídky jedna. Každá z položek menu je vlastně samostatný objekt.
- ✧ V inspektoru objektů si všimneme, že hlavní nabídka (komponenta MainMenu) má jen málo vlastností a žádné události.
- ✧ Mezi vlastnostmi položek nabídky, tedy komponent typu TMenuItem, najdeme Checked — zda se u položky objeví „zaškrtnutí“, Enabled — zda je položka aktivní nebo „zašeděná“, Visible — zda je za běhu programu viditelná atd. Vlastnost Radiobutton určuje, že ze skupiny položek můžeme vybrat vždy jen jednu.
- ✧ Separátor, tedy vodorovnou čáru oddělující skupinu položek v menu, vytvoříme jako položku, jejíž vlastnost Caption bude mít hodnotu „-“ (znak minus). I separátor je jednou z položek nabídky.
- ✧ Chceme-li vložit do nabídky novou položku jinam než na konec, označíme myší položku, před níž ji budeme vkládat, a stiskneme klávesu INS. Další postup je stejný jako při vkládání nové položky na konec nabídky. Chceme-li některou z položek nabídky odstranit, označíme ji a stiskneme klávesu DEL.
- ✧ Chceme-li vytvořit podřízenou nabídku, klikneme v editoru nabídek pravým tlačítkem myši na položku, která ji má vyvolat. Objeví se příruční menu, ve kterém zadáme příkaz Create Submenu. Editor menu se pak postará o vše potřebné. S položkami podřízené nabídky zacházíme stejně jako s položkami „obyčejné“ nabídky. Chceme-li odstranit podřízenou nabídku, smažeme všechny její položky.
- ✧ Vlastnost ShortCut umožňuje definovat pro danou položku nabídky klávesovou zkratku.
- ✧ Vytvořenou nabídku si můžeme uložit jako šablonu pro budoucí použití. K tomu poslouží příkaz z příručního menu Save As Template.... Hotovou šablonu použijeme tak, že kliknutím aktivujeme místo, kam ji chceme vložit, a z příručního menu vybereme příkaz Insert From Template....
- ✧ Můžeme také použít nabídku vytvořenou jiným nástrojem, pokud je popsána způsobem obvyklým pro programy pro Windows, tj. v souboru .RC. K tomu nám poslouží příkaz Insert From Resource... z příruční nabídky.

Pracujeme s nabídkou

Nyní naučíme svůj program reagovat na vybrání položky menu.

Ve vizuálním návrhu programu se již zobrazil pruh s nabídkou, a tak můžeme myší vybrat položku Nápis | Konec. Prostředí na to zareaguje tak, že vytvoří kostru handleru a přejde do textového editoru[10], kde doplníme tělo — volání metody Close().

Tento postup ale není nejlepší: Reakce na tuto položku nabídky má být stejná jako reakce na stisknutí tlačítka Konec, a proto bychom ji neměli programovat dvakrát. (Vždy je rozumné jednu operaci programovat jen jednou.) Naštěstí nám C++ Builder umožňuje použít stejný handler k ošetření několika událostí.



Obr. 2.9 K ošetření události spojené s menu použijeme už hotový handler

Nejprve v seznamu objektů v záhlaví inspektora objektů vyhledáme položku Konec1 a přejdeme na kartu Events; pro položku nabídky je k disposici jediná událost, OnClick. Všimněte si, že vedle prázdného políčka pro jméno handleru je tlačítko se šipkou dolů. Po jeho stisknutí se objeví seznam už hotových handlerů, které bychom mohli použít (obr. 2.9). Zde nabízí jedinou položku, konecClick, ale ta nám vyhovuje, neboť udělá přesně to co chceme — ukončí běh programu.

[10] Pokud handler již existuje, přejde Builder do jeho zdrojového kódu.

Dialogové okno O programu

Vybereme-li v nabídce položku Nápověda | O programu..., mělo by se objevit dialogové okno s informacemi o programu — jméno programu, číslo verze, jméno autora atd. Začneme tím, že si toto dialogové okno vytvoříme. V C++ Builderu je to okno jako každé jiné, liší se jen hodnotami některých vlastností a způsobem zobrazení.

Stisknutím tlačítka New Form na panelu nástrojů vytvoříme nové okno. (Můžeme také použít nabídku File | New a v zásobníku objektů (obr. 2.1) vybrat položku Form, obě možnosti znamenají totéž.) Tím se zároveň vytvořil zdrojový soubor s názvem Unit1.cpp, hlavičkový soubor Unit1.h a soubor Unit1.dfm s popisem nového okna a tyto soubory se připojily k projektu. Jestliže se nyní pokusíme uložit projekt (např. tlačítkem Save All), vyzve nás prostředí, abychom tyto soubory přejmenovali; použijeme např. jméno about.cpp; prostředí pojmenuje odpovídajícím způsobem i soubory .h a .dfm.

Toto okno pojmenujeme vtipně okno2 a do titulkové lišty (vlastnost Caption) napíšeme „O programu“. Dále nastavíme vlastnost BorderStyle na hodnotu bsDialog; tím zabezpečíme, že okno bude mít standardní rámeček dialogových oken, jeho velikost nepůjde měnit a v pravém horním rohu nebude mít tlačítka pro minimalizaci a maximalizaci.

Odstraníme ještě tlačítko pro uzavření okna (s křížkem). V inspektoru objektů vyhledáme vlastnost BorderIcons; nalevo od ní je křížek, který naznačuje, že jde o složenou vlastnost. Kliknutím na tento křížek se vlastnost „rozvine“ — pod ní se objeví její složky, dílčí vlastnosti. V našem případě to jsou informace, které z možných tlačítek se má zobrazit. Najdeme zde položku biSystemMenu a přidělíme jí hodnotu false. (Tato změna se projeví až za běhu programu.)

Všimněte si, že toto okno má vlastnost Visible implicitně nastavenu na false, tj. po spuštění programu se samo nezobrazí.

Dále na tento formulář umístíme komponentu Label a nastavíme její vlastnosti tak, aby zobrazovala vhodný nápis (to už umíme).

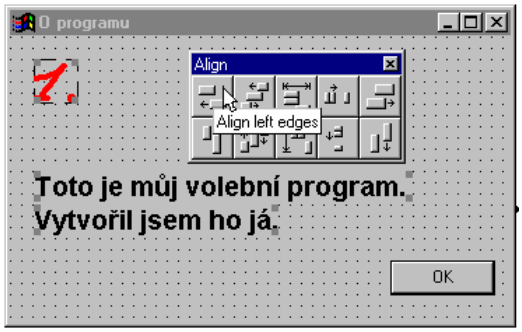
V okně O programu se také obvykle zobrazuje ikona (nebo jiný symbol) programu. Zobrazíme v něm tedy ikonu: Na paletě Additional vybereme komponentu Image, umístíme ji na vhodné místo, pojmenujeme ji ikona a vlastnost AutoSize nastavíme na true. (Velikost se automaticky přizpůsobí velikosti obrázku, který se bude zobrazovat. Pak vyhledáme vlastnost Picture a stiskneme tlačítko se třemi tečkami vpravo od její hodnoty. Otevře se dialogové okno, ve kterém si najdeme soubor s ikonou a otevřeme jej.

Nakonec přidáme tlačítko, které toto okno zavře. Tentokrát použijeme komponentu Button z palety Standard; pojmenujeme ji OKButton a napíšeme na ni „OK“. Abychom toto dialogové okno mohli zavřít také stisknutím kláves ESC nebo ENTER, nastavíme na true také vlastnosti Default a Cancel. (Komponenta Button se od BitBtn liší jen tím, že neobsahuje bitovou mapu.)

Pro snazší zarovnání jednotlivých komponent v dialogovém okně můžeme použít nástroj Alignment Palette, který vyvoláme z hlavní nabídky Builderu příkazem View | Alignment Palette (obr. 2.10). Obsahuje tlačítka, která umožňují vybrané prvky vystředit horizontálně i vertikálně, zarovnat podle levého nebo pravého okraje atd.

Nakonec napíšeme handler ošetřující stisknutí tlačítka OK, tj. událost OnClick. Jeho tělo bude obsahovat už známý příkaz — volání metody Close(). Tentokrát ovšem nemůžeme použít handler, který reagoval na stisknutí tlačítka Konec v hlavním okně, neboť zde voláme metodu třídu okno2 (a zavíráme jen dialogové okno).

Nyní zbývá jediné — vrátit se k hlavnímu oknu a doplnit handler, který po vybrání nabídky Nápověda | O programu... toto dialogové okno zobrazí. Příkazem View | Forms... nebo stisknutím tlačítka View Forms na panelu nástrojů si vyvoláme dialogové okno, v němž zvolíme okno1. Ve vizuálním návrhu okna zvolíme potřebnou položku menu, prostředí vytvoří kostru handleru Tokno1::Oprogramu1Click() a přenese nás do editoru do těla této funkce.



Obr. 2.10 Panel s nástroji pro vyrovnání komponent v dialogovém okně

Dialogové okno O programu se automaticky vytvoří[11] při spuštění programu, nebude však viditelné — jeho vlastnost Visible má hodnotu false. To znamená, že bychom mohli tělo handleru Tokno1::Oprogramu1Click() napsat např. takto:

```
void __fastcall Tokno1::Oprogramu1Click(TObject *Sender)
{
  okno2 -> Visible = true;     // Toto řešení není nejlepší
}
```

Zde prostě přiřadíme vlastnosti Visible hodnotu true a tím způsobíme, že se okno zobrazí. Stejného výsledku bychom dosáhli příkazem

```
okno2 -> Show();
```

(Show() je metoda zděděná od třídy TForm.)

Jestliže nyní program přeložíme a spustíme, poběží a bude se chovat na první pohled tak, jak očekáváme — ale jen na první pohled. Brzy zjistíme, že dialogové okno O programu může ztratit fokus, tj. že i když je na obrazovce, může být aktivní hlavní okno našeho programu. To není obvyklé; dialogová okna jsou většinou *modální*. Pokud je takové okno na obrazovce, vyžaduje, aby se mu uživatel věnoval, a nedovolí přejít do jiných oken aplikace, dokud je neuzavřeme. Použijeme tedy místo metody Show() metodu ShowModal():

```
void __fastcall Tokno1::Oprogramu1Click(TObject *Sender)
{
  okno2 -> ShowModal();
}
```

Nyní se program již chová tak, jak od něj očekáváme.

Poznámky

- ✧ Program v této podobě najdeme na CD v adresáři KAO02\02.
- ✧ Funkce ShowModal() vrací hodnotu, která indikuje, zda uživatel v daném okně stiskl tlačítko OK, Storno, nebo některé další. Zde tuto hodnotu nepotřebujeme, neboť uživatel nemá na vybranou, musí stisknout OK.
- ✧ Modální dialogové okno představuje vel;mi často žádost o informace, bez kterých program nemůže pokračovat. Protože může obsahovat více tlačítek, musíme u každého z nich předepsat, jaký výsledek jeho stisknutí znamená. K tomu slouží vlastnost ModalResult, která může nabývat některé z hodnot mrNone, mrOk, mrCancel, mrYes, mrNo, mrAbort, mrRetry a mrIgnore. V našem případě jsme tlačítku OK ponechali implicitní hodnotu mrNone, neboť nám na hodnotě vrácené funkcí ShowModal() nezáleží.
- ✧ Pokud přiřadíme vlastnosti ModalResult jinou hodnotu než mrNone, způsobí stisknutí tlačítka automaticky uzavření okna a vrácení odpovídajícího výsledku.

[11] Bude-li v programu větší množství oken, může automatické vytvoření všech hned při spuštění programu zbytečně zatěžovat systémové prostředky. Proto lze předepsat, která okna se mají vytvořit automaticky a která si chceme dynamicky vytvořit sami. Naučíme se to později.

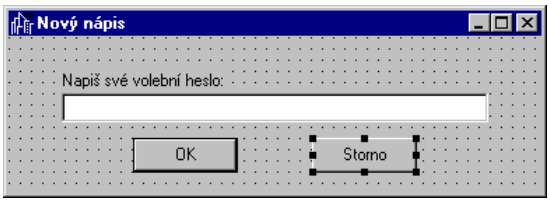
Měníme nápis

Náš program by také měl umět změnit zobrazovaný nápis — co kdybychom např. změnili politickou orientaci. K tomu použijeme dialogové okno s editačním polem.

Vytvoříme tedy nové okno, které nazveme vstup. Do titulku (vlastnost Caption) mu napíšeme „Nový nápis“, vlastnost BorderStyle nastavíme na bsDialog a pomocí vlastnosti BorderIcons potlačíme zobrazování systémového menu (to už také umíme). Pak ho uložíme do souboru novy.cpp. Na toto okno položíme komponentu Edit, která představuje editační pole, nazveme ji edit, upravíme její velikost a zarovnáme ji do středu okna. Nad ni umístíme komponentu Label, která ponese nápis „Napiš své volební heslo:“ Levý okraj nápisu zarovnáme s levým okrajem editačního pole.

Pod editační pole umístíme dvě tlačítka (komponenty Button). První z nich pojmenujeme OKButton, napíšeme na ně „OK“ a jeho vlastnost Default nastavíme na true. (Stisknutí klávesy ENTER bude ekvivalentní stisknutí tlačítka OK.) Pak ještě nastavíme vlastnost ModalResult na mrOk.

Druhé tlačítko nazveme Storno, napíšeme na ně „Storno“ a na true nastavíme vlastnost Cancel. (Stisknutí klávesy ESC bude ekvivalentní stisknutí tohoto tlačítka.) Nakonec nastavíme vlastnost ModalResult na mrCancel.



Obr. 2.11 Dialogové okno pro zadání nového nápisu

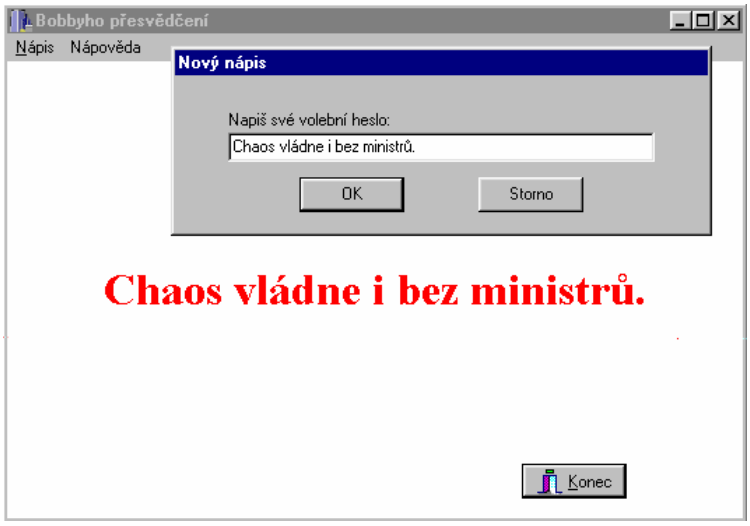
Toto okno vyvolá handler ošetřující vybrání položky menu Nápis | Nový.... Jeho text je jednoduchý:

```
void __fastcall Tokno1::NovyClick(TObject *Sender)
{
  if(vstup->ShowModal() == mrOk)
    heslo -> Caption = vstup -> edit -> Text;
}
```

Jestliže uživatel stiskne v našem dialogovém okně tlačítko OK, vrátí funkce ShowModal() kodnotu mrOK. V tom případě překopírujeme text z editačního pole do nápisu. (Nápis v editačním poli je hodnotou jeho vlastnosti Text.)

Poslední, co ještě musíme udělat, je zajistit, aby při otevření dialogového okna nabídlo editační pole text, který náš program právě zobrazuje. Protože dialogové okno při otevření vždy dostane fokus, použijeme událost OnActivate. Odpovídající handler prostě překopíruje text z komponenty heslo z hlavního okna do vlastnosti Text editačního pole:

```
void __fastcall Tvstup::FormActivate(TObject *Sender)
{
  edit->Text = okno1 -> heslo -> Caption;
}
```



Obr. 2.12 Konečná podoba našeho prvního programu

To je vše. Všimnete si, že jsme nedefinovali handlery pro stisknutí tlačítek OK a Storno. To není třeba, neboť jsme pro tato tlačítka definovali vlastnost ModalResult.

Obrázek 2.12 ukazuje výsledek našeho snažení. Poslední verzi programu najdeme va CD v adresáři KAP02\03.

Šíření programu

Svůj program jsme zdárně dokončili, je na čase ho začít šířit mezi lidi. K tomu účelu ovšem nejprve vytvoříme „ostrou“ verzi. (Při implicitním nastavení vytváří C++ Builder ladicí verzi bez optimalizací a s ladicími informacemi v přeloženém programu.) Příkazem Project | Options... vyvoláme dialogové okno Project Options a na kartě Compiler stiskneme tlačítko Release.

Dále se musíme rozhodnout, zda budeme svůj program dodávat jako jeden jediný soubor nebo jako spustitelný soubor spolu s několika dynamickými knihovnami. Druhá možnost zpravidla vyžaduje použití instalačního programu, a to je pro předvolební program nevhodné. Proto přejdeme na kartu Linker a zrušíme zaškrtnutí políčka Use dynamic RTL (použít dynamickou verzi knihovny jazyka C++); pak přejdeme na kartu Packages a zrušíme zaškrtnutí políčka Build with runtime packages. (Packages, tedy „balíčky“, jsou dynamické knihovny obsahující komponenty; mají příponu .BPL).

Po přeložení můžeme svůj program (tj. soubor hello2.exe) již bez problémů šířit.

2.3 Komponenty

Hlavní síla C++ Builderu spočívá ve využití komponent. Co to vlastně je?

Komponenty jsou předdefinované a speciálním způsobem přeložené třídy, které zapouzdřují relativně samostatné a univerzálně použitelné součásti programů — ovládací prvky z Windows (tlačítka, editační pole atd.), nástroje pro přístup k databázím, systémové nástroje (časovač) atd. Také okna — „obyčejná“ i dialogová — jsou instance vizuálních komponent. Na rozdíl od běžných tříd v C++ však rozhraní komponent tvoří především *vlastnosti* a *události*. Navíc jsou některé komponenty přeloženy tak, že je lze instalovat na paletách a vkládat pomocí myši do vizuálního návrhu aplikace. U takto vytvořených instancí pak můžeme určovat hodnoty vlastností už v době návrhu pomocí inspektora objektů. (Vedle toho ovšem existuje řada komponent, které tímto způsobem nelze používat.)

Vlastnosti

Vlastnosti (*properties*) na první pohled připomínají datové složky (atributy) instancí, jejich používání se ale liší. Tím, že změníme hodnotu vlastnosti, se ihned změní vizuální či jiné vlastnosti instance. Např. okna, tlačítka a mnohé další komponenty mají vlastnost Visible. Chceme-li komponentu zobrazit, stačí k tomu přiřazení

```
Visible = true;
```

Podobně ke skrytí komponenty stačí příkaz

```
Visible = false;
```

Pokud bychom chtěli něco podobného naprogramovat bez pomoci komponent, museli bychom nejspíš definovat vlastnost Visible jako třídu a přetížit pro ni operátor = tak, že bude volat funkci ::ShowWindow() z aplikačního rozhraní Windows.

Nastavení hodnoty vlastnosti — např. přiřazením — může způsobit volání určité funkce; zpravidla jde o chráněnou metodu třídy komponenty. Podobně i použití hodnoty vlastnosti může způsobit volání určité metody. Hodnota vlastnosti může, ale nemusí být uložena v datové složce komponenty.

Hodnoty některých vlastností můžeme nastavovat už v době návrhu pomocí inspektora objektů, hodnoty jiných musíme určovat až za běhu programu.

Deklarace vlastnosti

K deklaracím komponent se dostaneme později. Přesto si už teď ukážeme, jak se vlastnosti deklarují (především abychom snáze porozuměli nápovědě).

Deklarace vlastnosti vypadá podobně jako deklarace datové složky, je však uvedena klíčovým slovem __property. Za identifikátorem vlastnosti pak následuje znak = a za ním ve složených závorkách specifikace metod, které se použijí pro nastavení (write) resp. zjištění (read) hodnoty vlastnosti. Může tu také být uveden identifikátor datové složky, ve které je hodnota vlastnosti uložena. Dále tu může být specifikována implicitní hodnota vlastnosti (default, resp. nodefault, pokud vlastnost nemá implicitní hodnotu) a zda se daná vlastnost ukládá při uložení komponenty na disk (stored). Syntax ukazuje následující příklad.

Vlastnost Visible, která je společná pro mnoho tříd komponent a která určuje, zda bude odpovídající okno nebo jinou složku grafického rozhraní programu vidět, je deklarována takto:

```
__property bool Visible = {read=FVisible, write=SetVisible, default=1};
```

Tato deklarace říká, že Visible je vlastnost typu bool, že přiřadíme-li jí hodnotu, zavolá se metoda SetVisible(), použijeme-li její hodnotu, vezme se hodnota uložená v datové složce FVisible, a že implicitní hodnota této vlastnosti je 1 (true).

Poznamenejme, že metody, které se používají pro nastavení hodnoty vlastnosti Xyz, mají obvykle jméno SetXyz(), metody pro zjištění hodnoty této vlastnosti mají obvykle jméno GetXyz() a proměnná, do níž se hodnota vlastnosti ukládá, mívá jméno FXyz.

Některé vlastnosti komponent

Name	Mezi vlastnostmi komponent instalovaných na paletách má zvláštní postavení — určuje identifikátor instance ve zdrojovém textu programu. Hodnotu této komponenty můžeme měnit pouze v době návrhu.
Align	Určuje, zda se komponenta zarovná k některému z okrajů panelu nebo okna.
Anchors	Určují chování komponenty při změně velikosti okna. (Jen ve verzi 4.)
Color	Určuje barvu pozadí komponenty.
Constraints	Umožňuje stanovit omezení pro rozměry komponenty. (Jen ve verzi 4.)
Cursor	Určuje tvar kurzoru při přechodu přes danou komponentu.
Enabled	Určuje, zda komponenta může být aktivní (dostat fokus).
Font	Skládá se z řady dílčích vlastností, které určují charakteristiky písma: druh, barvu, řez, velikost atd.
Hint	Obsahuje text „bublinové nápovědy“, která se zobrazí, když uživatel ponechá nějakou dobu kurzor myši na komponentě, a případně nápovědy, která se zobrazí ve stavovém řádku. Tyto dvě části musí být odděleny znakem „|“.

ShowHint	Určuje, zda se bude bublinová nápověda a nápověda ve stavovém řádku zobrazovat. Aby se zobrazila, musí mít hodnotu true vlarstnost ShowHint komponenty a objektu Application.
Tag	Tato vlastnost představuje proměnnou typu int, kterou můžeme využít k vlastním cílům.
ParentFont, ParentColor, ParentCtl3D, ParentShowHint	Tyto vlastnosti určují, zda komponenta převezme vlastnost Font, Color, Ctl3D nebo ShowHint od svého „rodiče“, tj. od okna nebo panelu, do něhož je vnořena. Tak stačí např. nastavit Font pro okno a převezmou ho všechny komponenty, které na něm použijeme a pro jejichž vlastnost ParentFont bude mít hodnotu true.
Left, Top	Relativní souřadnice levého horního rohu komponenty vzhledem k oknu (v pixelech)
Height, Width	Svislý, resp. vodorovný rozměr komponenty (v pixelech)
ClientHeight, ClientWidth	Svislý, resp. vodorovný rozměr klientské oblasti okna (tedy oblasti, do které běžně kreslíme)

Události

Každá komponenta může reagovat na určité události; jejich přehled pro danou komponentu najdeme v okně inspektora objektů na kartě Events. Jejich jména většinou začínají On..., např. OnClick, OnDblClick atd. Komponenta na tyto události reagovat může, ale nemusí.

Reakci komponenty na událost obstarává handler. Podívejme se znovu na deklaraci třídy okno2 z našeho prvního příkladu (soubory about.h a about.cpp).

```
class Tokno2 : public TForm
{
__published:
   TLabel      *napis1;
   TImage      *ikona;
   TButton     *OKbutton;
   TLabel      *napis2;
   void __fastcall OKbuttonClick(TObject *Sender);
private:
public:
   __fastcall Tokno2(TComponent* Owner);
};
__fastcall Tokno2::Tokno2(TComponent* Owner)
   : TForm(Owner)
{
}
void __fastcall Tokno2::OKbuttonClick(TObject *Sender)
{
  Close();
}
```

Schéma je jednoduché: Třída okna je odvozena od společného předka, třídy TForm. Tato třída slouží jako kontejner, který obsahuje ukazatele na další komponenty a handlery pro obsluhu událostí, na které mohou komponenty reagovat.

Handler je metoda okna obsahujícího komponentu, nikoli metoda komponenty samotné. Díky tomu můžeme jeden handler použít k obsluze podobných událostí u různých komponent.

Parametrem handleru je ukazatel Sender na komponentu, které se událost týká. Je typu TObject*, kde TObject je společný předek všech tříd z knihovny VCL. V případě potřeby ho můžeme přetypovat na ukazatel na skutečný typ komponenty, nejlépe pomocí operátoru dynamic_cast; hodí se, chceme-li použít jeden handler pro více událostí.

Poznamenejme, že handlery pro některé události mohou mít ještě další parametry; parametr Sender ale mají vždy.

Některé běžné události

S alespoň některými z následujících událostí se setkáme u většiny komponent.

OnClick	Nastane, když klikneme na komponentě myší.
OnDblClick	Nastane při dvojkliku myší na dané komponentě.
OnEnter	Nastane v okamžiku aktivace (když komponenta dostane fokus).
OnExit	Nastane v okamžiku, kdy komponenta ztratí fokus.
OnMouseDown	Nastane při stisknutí tlačítka myši, je-li kurzor na komponentě.
OnMouseUp	Nastane při uvolnění tlačítka myši, je-li kurzor na komponentě.
OnMouseMove	Nastane při pohybu myši přes komponentu. Přitom nezáleží na tom, zda je nebo není stisknuto některé z tlačítek.
OnPaint	Nastane vždy, když je třeba komponentu překreslit. (Při práci s vizuálními komponentami ji téměř nepotřebujeme.)
OnChange	Nastane, když se komponenta nějakým způsobem změní; podrobnosti se u různých komponent liší.
OnKeyDown	Nastane, je-li komponenta aktivní (má-li fokus) a stiskneme-li přitom některou z kláves.
OnKeyUp	Nastane, je-li komponenta aktivní (má-li fokus) a uvolníme-li přitom některou z kláves.

Z dodatečných parametrů handlerů pro události vázané na myš lze zjistit polohu kurzoru myši, které tlačítko bylo stisknuto, zda byl zároveň stisknut některý z přeřaďovačů SHIFT, CTRL atd.

Pohled do pozadí

Komponenty volají události prostřednictvím ukazatelů. Jestliže např. komponenta TButton může reagovat na událost OnClick, znamená to, že každá instance třídy TButton obsahuje datovou složku OnClick. Tato složka může obsahovat ukazatel[12] na handler.

Nastane-li odpovídající událost, podívá se program, zda tento ukazatel obsahuje adresu handleru, a pokud ano, zavolá ho; jinak se nic nestane.

To znamená, že handlery pro různé události můžeme měnit i za běhu programu.

Metody

Podívejme se nyní na některé užitečné metody komponent.

ClientToScreen()	Převede souřadnice relativní vzhledem k oknu na souřadnice vzhledem k celé obrazovce.
Hide()	Skryje komponentu. Komponenta dále existuje a může být později zobrazena, např. metodou Show().
Invalidate()	Požádá o překreslení komponenty. K tomu dojde, jakmile se to bude operačnímu systému hodit.
Refresh()	Vymaže pozadí komponenty a způsobí okamžité překreslení.
Repaint()	Způsobí okamžité překreslení komponenty, ale nevymaže pozadí.
SetBounds()	Umožňuje nastavit najednou (jedním voláním) vlastnosti Top, Left, Width a Height.
SetFocus()	Přenese fokus na zadanou komponentu.

[12] Pozor, nejde o tradiční ukazatel na metodu („třídní ukazatel“), jak jej známe z C++, ale o ukazatel zvláštního typu __closure. Podrobněji viz kap. 11.

Knihovna VCL

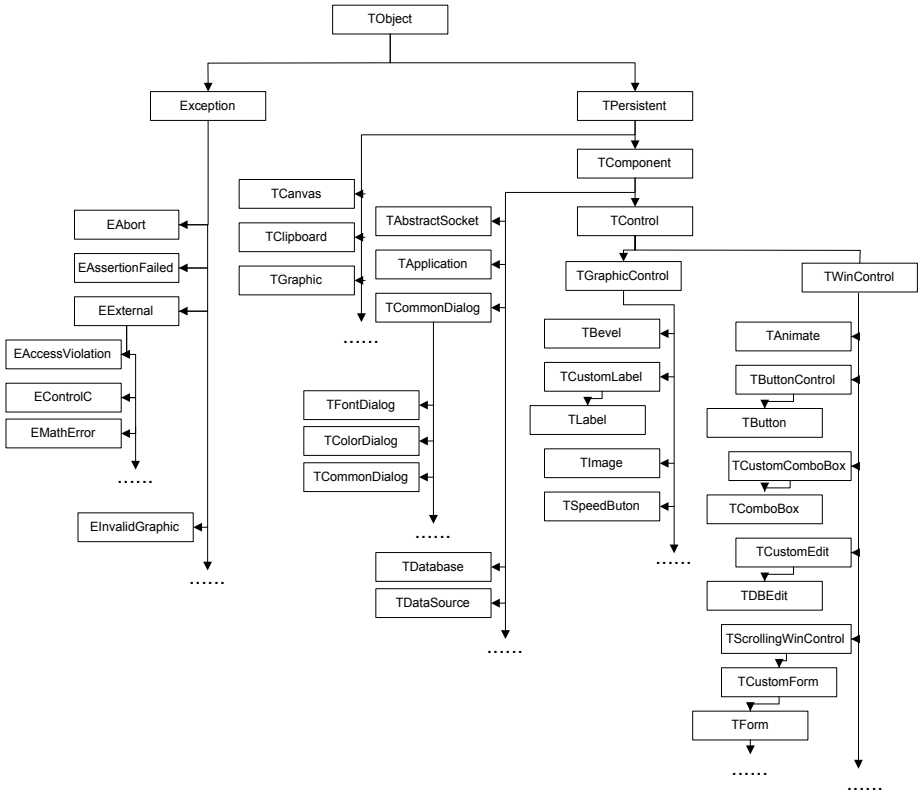
Komponenty v Borland C++ Builderu jsou uloženy v knihovně *Visual Component Library* (*VCL*). Tvoří hierarchii objektových typů, jejíž část ukazuje obr. 2.

Tuto knihovnu C++ Builder sdílí s Delphi. *Všechny standardní komponenty jsou napsány v Object Pascalu.* Z toho plynou některá omezení:

- ✧ Všechny komponenty mají společného předka — třídu TObject (přesněji: komponenty mají za předka třídu TComponent, která je odvozena od TObject).
- ✧ Veškeré instance tříd z VCL a jejich potomků musíme alokovat dynamicky (pomocí operátoru new) a pracovat s nimi pomocí ukazatelů.
- ✧ Při odvozování nových komponent nelze využívat vícenásobné dědičnosti.
- ✧ Ve starších verzích Builderu nebylo možno přetěžovat metody komponent, takže máme např. vždy jen jeden konstruktor.
- ✧ Chceme-li programovat komponenty, musíme se naučit některá rozšíření jazyka C++ (viz kap. $$$).
- ✧ Při používání komponent se budeme muset seznámit s některými zvláštními datovými typy (AnsiString pro práci s řetězci ap.)

Podívejme se nyní na některé důležité třídy z této hierarchie.

TObject	Společný předek všech prvků této knihovny
TApplication	Třída, jejíž instance obsahují základní funkčnost okenní aplikace; obsahuje např. seznam oken, stará se zpracování zpráv od Windows atd.
TComponent	Společný předek všech komponent (vizuálních i nevizuálních)
TControl	Společný předek vizuálních komponent
TForm	Okno (formulář), tj. předek, od kterého jsou odvozeny třídy oken v programech
Exception	Společný předek tříd pro přenos informací o výjimkách (tedy pro klasifikaci chyb)
TCanvas	Třída vyjadřující kreslicí plochu komponent; zapouzdřuje „kontext zařízení“ známý z programování ve Windows API



Obr. 2.11 Část hierarchie objektových typů tvořících knihovnu VCL

Komponenty na paletách

Na závěr této kapitoly si uděláme velmi stručný přehled palet a komponent na nich instalovaných. Jména tříd těchto komponent jsou tvořena jménem komponenty, před které doplníme písmeno T.

Paleta Standard

Na této paletě najdeme především běžné prvky uživatelského rozhraní programů — nabídky (komponenty MainMenu a PopupMenu), text vložený do okna (Label), editovací textová pole (Edit, Memo), běžné ovládací prvky uživatelského rozhraní programů pro Windows (Button, Checkbox, RadioButton, ListBox, ComboBox, Scrollbar), komponenty určující skupiny prvků (GroupBox, RadioGroupBox). Komponenta Panel umožňuje snazší zarovnání dalších prvků.

Paleta Additional

Tato paleta obsahuje další běžné ovládací prvky. Najdeme tu např. komponentu Image pro kreslení nebo zobrazení bitových map, Chart pro vytváření grafů, BitBtn pro tlačítko s obrázkem, Splitter pro rozdělení okna, MaskEdit pro implementaci editačního pole, které kontroluje vstupní data, komponentu CheckListBox, která představuje seznam se zaškrtávacími políčky atd.

Paleta Win32

Zde jsou soustředěny komponenty zapouzdřující ovládací prvky z „nového uživatelského rozhraní“ Win32 jako RichEdit (vlastně malý textový editor), Animate pro animaci, strom (TreeView), seznam obrázků (ImageList), stavová řádka (StatusBar), panel nástrojů (ToolBar), nástroje pro vícestránková dialogová okna (PageControl, TabControl) atd.

Paleta System

Na této paletě najdeme vedle časovače (Timer) komponentu MediaPlayer, která umožňuje řídit zařízení jako CD-ROM nebo MIDI sequencer. Komponenta PaintBox umožňuje kreslení (podobně jako Image). Další komponenty na této paletě podporují OLE a DDE.

Paleta Internet

Na této paletě najdeme komponenty, které podporují programování pro sítě. Lze je rozdělit do dvou skupin: „Nízkoúrovňové“ komponenty zapouzdřují rozhraní Windows Sockets (ClientSocket, ServerSocket a další), „vysokoúrovňové“ komponenty obsahují nástroje pro práci s internetem nebo intranetem (HTML, NMHTTP, NMFTP a další).

Paleta Data Access

Obsahuje komponenty pro přístup k datům v databázových aplikacích — DataSource (zdroj dat), Table (databázová tabulka), Query (dotaz), StoredProc (uložená procedura) atd.

Paleta DataControls

Zde najdeme komponenty pro zobrazování dat z databází: DBGrid (tabulka pro zobrazení dat), DBNavigator (ovládací prvek pro přechody mezi záznamy), DBCheckBox a DBRadioGroup pro zobrazování dat ve tvaru voleb, DBEdit pro zobrazování textových údajů, DBChart pro vytváření grafů atd.

Paleta QReport

Na této paletě jsou soustředěny komponenty pro vytváření sestav, tedy výpisů z databází, pomocí nástroje Quickreport.

Paleta Dialogs

Obsahuje předdefinovaná dialogová okna pro volba souboru, který chceme uložit nebo přečíst, pro výběr barvy, fontu, uložení nebo načtení obrázku, nastavení nebo volbu tiskárny atd.

Paleta Win 3.1

Obsahuje některé další ovládací prvky uživatelského rozhraní, např. DBLookupList, DBLookupCombo, TabSet, TabbedNotebook atd.

Paleta ActiveX

V C++ Builderu lze podobně jako komponenty používat i prvky ActiveX. Na této paletě najdeme např. Graph.

Paleta Decision Cube

Obsahuje nástroje pro vytváření aplikací pro podporu rozhodování, které umožňují vytvářet přehledy z velkého množství dat. Najdeme tu komponenty DecisionCube, DcisionQuery, DecisionGraph a další. Najdeme ji pouze ve verzi Enterprise.

Paleta Midas

Obsahuje komponenty pro vytváření vícevrstvých databázových aplikací. Najdeme ji jen ve verzi Enterprise..

Použití komponent na paletách

Chceme-li vložit do programu jednu komponentu z palety, vybereme ji tím, že klikneme na její symbol na paletě, a pak ji vložíme kliknutím na požadované místo ve vizuálním návrhu nebo v datovém modulu (o tom budeme hovořit později). Tím se zároveň vytvoří odpovídající kód.

Výběr komponenty zrušíme buď výběrem jiné komponenty nebo kliknutím na šipku, která je na paletě úplně vlevo.

Chceme-li do vizuálního návrhu programu vložit několik instancí jedné komponenty, stiskneme při výběru zároveň tlačítko SHIFT. Každé následující kliknutí do vizuálního návrhu pak způsobí vložení jedné instance této komponenty. Poté obvyklým způsobem zrušíme výběr komponenty.

Chceme-li komponentu z programu odstranit, vybereme ji ve vizuálním návrhu a stiskneme klávesu DEL.

3. Hodiny

V této kapitole napíšeme program, který bude ukazovat přesný čas podle počítače. To nám umožní seznámit se s časovačem (timer). Tato komponenta — je-li spuštěna, tj. má-li její vlastnost Enabled hodnotu true — vyvolává v pravidelných intervalech událost OnTimer a tak umožňuje pracovat v programech s časem. Mimo to se naučíme kreslit do okna.

3.1 Digitálky

Začneme digitálními hodinami, neboť ty jsou jednodušší. Program bude prostě zobrazovat znakový řetězec vyjadřující aktuální čas.

Čas zjistíme pomocí globální funkce Time(), která vrací aktuální datum a čas uložené v instanci třídy TDateTime. Tento údaj převedeme na znakový řetězec pomocí funkce TimeToStr(), která vrátí hodnotu typu AnsiString[13]. Výsledný řetězec odpovídá místním zvyklostem nastaveným v operačním systému.

Vytvoříme tedy nový projekt a uložíme ho pod názvem hodiny1. Hlavní okno pojmenujeme hodinky a do jeho titulku vložíme nápis „Přesný čas“.

Ke zobrazení času bychom mohli použít již známou komponentu Label. Stejně dobře však poslouží i komponenta Panel, kterou najdeme také na paletě Standard. Panel je ve skutečnosti kontejner na komponenty, může ale uprostřed zobrazovat nápis, jehož text je určen vlastností Caption. Vložíme tedy do okna komponentu Panel, kterou pojmenujeme cas, neboť bude zobrazovat časový údaj. Její vlastnost Align nastavíme na hodnotu alClient, takže zabere celou klientskou oblast okna a při změně velikosti okna se přizpůsobí.

Pak smažeme hodnotu vlastnosti Caption, neboť zobrazovaný text — přesný čas — budeme znát až za běhu programu. Nakonec stanovíme druh písma pomocí vlastnosti Font. Použijeme např. písmo Arial o velikosti 60 bodů.

Dále umístíme (kamkoli do okna) komponentu Timer, kterou pojmenujeme nepokoj, neboť zde bude hrát podobnou roli jako nepokoj ve mechanických hodinkách. Tato komponenta není vizuální (za běhu programu ji není vidět), takže na přesném umístění nezáleží (podobně jako u komponenty MainMenu).

Časovač vyvolává v pravidelných intervalech událost OnTimer. Délku tohoto intervalu v milisekundách určuje vlastnost Interval. Znakový řetězec udávající čas se bude měnit jednou za sekundu, a proto přidělíme této vlastnosti hodnotu 1000.

Nakonec vytvoříme handler, který bude událost OnTimer ošetřovat. Jeho zdrojový text bude jednoduchý:

```
void __fastcall Thodinky::nepokojTimer(TObject *Sender)
{
    cas -> Caption = TimeToStr(Time());
}
```

Nyní si můžeme program vyzkoušet. Brzy zjistíme, že má jednu nevýhodu: Jestliže příliš zmenšíme jeho okno, nebude text v něm vidět celý. Bylo by rozumné, kdyby se velikost číslic měnila v závislosti na rozměrech okna (alespoň v nějaké míře).

Změníme-li rozměry okna, způsobíme událost OnResize; napíšeme tedy handler, který na ni bude reagovat. V něm nejprve pomocí vlastností Height a Width zjistíme novou velikost okna v pixelech a z ní pak vypočítáme novou velikost písma např. jako jednu čtvrtinu menšího z rozměrů. Protože ale příliš velké nebo příliš malé písmo není příjemné, omezíme jeho velikost na rozmezí od 4 do 300 pixelů.

Velikost písma se tradičně udává v typografických bodech. Vlastnost Font komponenty Panel (a nejen této komponenty) je však založena na třídě TFont, která naštěstí obsahuje vlastnost Height umožňující zadat velikost písma v pixelech. To nám ušetří přepočet.

Handler pro ošetření události OnResize může vypadat takto:

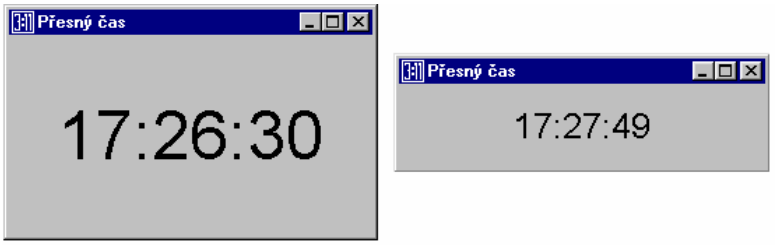
```
inline int min(int a, int b) {return a < b ? a : b;}
```

[13] AnsiString je třída používaná ve VCL pro práci se znakovými řetězci. Můžeme také používat označení String, které je zavedeno deklarací typedef v jednom z hlavičkových souborů knihovny VCL. Pozor, tato třída se liší od třídy string ze standardu C++.

```
void __fastcall Thodinky::FormResize(TObject *Sender)
{
  int vel = min(cas -> Height, cas -> Width)/4;      // Velikost písma v pixelech
  if(vel > 300) vel = 300;                           // Omezíme velikost na interval
  else if(vel < 4) vel = 4;                          // 4 - 300 pixelů
  cas -> Font->Height = vel;                              // Nastavíme velikost písma
}
```

Aby mohly hodiny plnit svůj účel, měly by být pořád na vrchu, nemělo by být možné zakrýt je jiným oknem. I to je jednoduché — stačí vzít vlastnost FormStyle okna a nastavit ji na hodnotu fsStayOnTop.

Zbývá vytvořit ikonu a připojit ji k programu; to přenechávám vaší iniciativě. Výsledek ukazuje obr. 3.1. (Hotový program i s ikonou najdete na CD v adresáři KAP03\01.)



Obr. 3.1 Přesný čas při různém tvaru a velikosti okna

Časovač

Zastavme se nyní na chvilku u časovače. Komponenta Timer, tedy instance třídy TTimer, zapouzdřuje časovač z Windows API. Časovač ve Windows — je-li spuštěn — ukládá v pravidelných intervalech do fronty zprávu WM_TIMER. Náš program si pak musí tuto zprávu z fronty vyzvednout a zpracovat ji. (Program v C++ Builderu na ni reaguje tak, že vyvolá událost OnTimer. Pokud existuje handler pro tuto událost, zavolá se.)

Protože zpráva WM_TIMER obsahuje mj. údaj o čase, kdy byla uložena do fronty, mohlo by se zdát, že stačí tento údaj vzít a zobrazit. Ovšem handler, který událost OnTimer ošetřuje, nemá odpovídající parametr.

Při práci s časovačem (a nejen v Builderu nebo v Delphi, ale i ve Windows API) můžeme narazit i na další problémy, které vznikají ze způsobu, jakým Windows se zprávami WM_TIMER zacházejí.

- ✧ Mezi okamžikem, kdy zpráva přijde do fronty, a okamžikem, kdy si ji program z fronty vyzvedne, může uplynout různě dlouhá doba. Pokud program např. zpracovává náročný výpočet, může to být několik minut nebo ještě déle.
- ✧ Přijde-li do fronty nová zpráva WM_TIMER a ve frontě už nějaká zpráva WM_TIMER pro náš program je, nezařadí se nová zpráva na konec, ale nahradí zprávu předchozí. Ve frontě je tedy vždy nejvýše jedna zpráva WM_TIMER.
- ✧ Interval časovače lze nastavit na hodnoty od 1 milisekundy výše. Ve skutečnosti je však nejmenší rozumný interval 55 ms a skutečné intervaly jsou násobky této hodnoty; to je dáno implicitním nastavením časovače procesoru.

Okno bez titulku

Naše první hodiny fungují, je ale jasné, že k dokonalosti mají ještě hodně daleko. První, co nám zde může vadit, je titulková lišta; sdělení o tom, že jde o přesný čas, vyvolává pochybnosti o autorově inteligenci, mimo jiné už proto, že jde o čas nastavený v počítači, a ten nemusí mít s přesným časem nic společného. (Navíc pokud si nějaká aplikace zabere na delší dobu čas procesoru pro sebe, např. proto, že použije thread s vyšší prioritou, budou hodiny chvíli stát.) Ale ať na ni umístíme jakýkoli nápis, bude rušit.

Pokusíme se tedy titulkovou lištu odstranit.

Jednou z možností je nastavit vlastnost okna BorderStyle na hodnotu bsNone; výsledkem ovšem bude okno, které nejenže nebude mít titulkovou lištu, ale ani okraje nebo systémové menu. Takové okno nepůjde přemísťovat.

C++ Builder nabízí lepší řešení — předefinovat parametry, které se uplatňují při vytváření okna. Tyto parametry vytváří virtuální metoda CreateParams(TCreateParams &Par), která k tomu využije především vlastností okna definovaných pomocí inspektora objektů. Stačí tedy tuto metodu ve třídě okna předefinovat.

Parametr Par je struktura, která obsahuje mj. složku Style. Pro běžná okna se jako styl používá příznak WS_OVERLAPPEDWINDOW, který říká, že jde o okno s titulkovou lištou, systémovým menu[14] atd. My místo toho použijeme příznaky WS_POPUP (okno bez titulkové lišty, většinou se používá jako pomocné), WS_THICKFRAME (bude mít rám, který umožňuje měnit velikost okna) a WS_SYSMENU (bude mít systémové menu, které sice nebude vidět, ale které půjde vyvolat stisknutím klávesové zkratky ALT+MEZERA).

Naše funkce CreateParams() nejprve zavolá metodu CreateParams() předka; tato metoda vytvoří hodnoty všech parametrů. Teprve pak změníme hodnoty parametru Style. Definice třídy Thodinky tedy bude mít tvar

```
class Thodinky : public TForm
{
__published:
  TPanel *cas;
  TTimer *nepokoj;
  void __fastcall nepokojTimer(TObject *Sender);
  void __fastcall FormResize(TObject *Sender);
private:
public:
  __fastcall Thodinky(TComponent* Owner);
  void __fastcall CreateParams(TCreateParams &Par);  // Předefinujeme
};                                                   // virtu ln  metodu p edka
```

Definice třídy Thodinky

Poznamenejme, že specifikaci virtual smíme vynechat, překladač C++ si ji doplní automaticky. Definice metody CreateParams() bude

```
void __fastcall Thodinky::CreateParams(TCreateParams &Par)
{
 TForm::CreateParams(Par);      // Nejprve zavoláme metodu předka
 Par.Style = WS_POPUP | WS_THICKFRAME | WS_SYSMENU;
}
```

Předefinovaná metoda CreateParams()

Výsledek ukazuje obr. 3.2. Tento program můžeme ukončit buď tradiční klávesovou kombinací ALT+F4 (je-li okno s hodinami aktivní) nebo pomocí pruhu úloh ve Windows. Můžeme také klávesovou kombinací ALT+MEZERA vyvolat systémové menu a v něm zvolit položku Close. Velikost okna můžeme měnit obvyklým způsobem myší; k přemístění použijeme příkaz Move ze systémové nabídky a kurzorové klávesy.

Obr. 3.2 Hodiny, tentokrát bez titulkové lišty

[14] Příznak WS_OVERLAPPEDWINDOW je ve skutečnosti kombinací několika jednodušších příznaků. V hlavičkovém souboru windows.h je definován jako WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU | WS_THICKFRAME | WS_MINIMIZEBOX | WS_MAXIMIZEBOX. V našem případě je důležitá změna WS_OVERLAPPED na WS_POPUP a odstranění příznaku WS_CAPTION.

Velikost písma

Nyní se pokusíme vylepšit práci programu s velikostí písma. Implicitní nastavení způsobí, že při prvním spuštění velikost neodpovídá hodnotám, které bude mít později. Navíc hodnotu $^1/_4$ výšky okna jsme určili zkusmo tak, aby se časový údaj vešel celý do okna, ale v mnoha případech je výsledkem velké okno s malým nápisem.

Výšku písma známe, potřebujeme však určit délku nápisu.

Nejprve přesuneme výpočet výšky písma do samostatné metody ZjistiVelikostPisma(). To znamená, že handler, který zpracovává událost OnResize, bude mít nyní tvar

```
void __fastcall Thodinky::FormResize(TObject *Sender)
{
  int vel = ZjistiVelikostPisma();
  if (vel < 4) vel = 4;
  else if (vel > 300) vel = 300;
  cas -> Font -> Height = vel;
}
```

Nyní musíme určit velikost písma. K tomu nám poslouží metoda TextWidth() třídy TCanvas; použijeme např. vlastnost Canvas okna hodinky. (O komponentě Canvas bude podrobněji hovořit v následujícím oddílu.) Funkce ZjistiVelikostPisma() bude

```
int Thodinky::ZjistiVelikostPisma()
{
  hodinky -> Canvas -> Font -> Height = 100;                    // 1
  int sirka = hodinky -> Canvas -> TextWidth(TimeToStr(Time()));   // 2
  return min(cas ->Height, 100 * cas -> Width/sirka)* 7/8;  // 3
}
```

V prvním řádku nastavíme velikost písma na 100 pixelů. (Protože jde o velikost písma v okně, nikoli na panelu cas, a protože panel cas má vlastnost ParentFont nastavenu na false, tato dočasná změna se na zobrazení času neprojeví.)

Ve druhém řádku zavoláme metodu Canvas -> TextWidth() a jako parametr jí předáme aktuální čas převedený na znakový řetězec. Výsledek uložíme do proměnné sirka.

Ve třetím řádku vezmeme výšku okna a porovnáme ji s výškou písma, které by zcela vyplnilo šířku okna. (K výpočtu použijeme trojčlenku.) Z těchto hodnot vezmeme menší a výsledek vynásobíme $^7/_8$, takže písmo téměř vyplní celé okno.

Zbývá vyřešit počáteční nastavení velikosti písma. K tomu použijeme událost OnShow, která nastane vždy, když se okno zobrazuje. K ošetření této události použijeme stejný handler jako k ošetření události OnResize. Nyní bude okno obsahovat od počátku písmo správné velikosti.

Příruční menu

Ovládání programu ze systémové nabídky je poněkud nepohodlné, a proto k němu přidáme příruční menu vyvolávané stisknutím pravého tlačítka myši.

Na paletě Standard vyhledáme komponentu PopupMenu a vložíme ji kamkoli na formulář; pojmenujeme ji třeba menu. Dvojklikem na ni pak vyvoláme editor nabídek, který v tomto případě nabídne přímo první položku podřízené nabídky. Tuto položku uložíme pod názvem konec (vlastnost name) a napíšeme na ni Konec (vlastnost Caption). Dvojklikem na tuto položku v návrháři menu se vytvoří kostra handleru Thodinky::konecClick(); doplníme do ní příkaz Close().

Pak v inspektoru objektů vyhledáme okno hodinky a jeho vlastnost PopupMenu nastavíme na jméno našeho příručního menu, tj. menu (rozbalovací seznam nám ani jinou možnost nenabídne). Tím příruční nabídku „připojíme k oknu“, tj. zajistíme, že se po stisknutí pravého tlačítka myši objeví.

Obr. 3.3 Naše hodiny s upravenou velikostí písma a s příruční nabídkou

Co dál

Nyní můžete zkusit program vylepšovat sami — třeba pokud jde o vzhled. Zajímavých efektů lze dosáhnout různými kombinacemi vlastností BevelInner a BevelOuter (vnitřní a vnější sešikmení) panelu, stylů a šířky okrajů okna atd. Můžete také zkusit přidat do nabídky příkaz Stop, který hodiny zastaví, a příkaz Pokračuj, který je opět spustí (nastaví vlastnost časovače Enabled na false, resp. true). Možný výsledek ukazuje obr. 3.3; hotový program najdete na doprovodném CD v adresáři KAP03\02.

3.2 Opravdové hodiny

Digitální hodiny jsou sice hezká věc, ale obávám se, že patří spíše do laboratoře; mají-li se mi hodiny líbit, musí mít ciferník a ručičky. Naprogramujme si je. Než se do toho však pustíme, musíme si ujasnit, jak se v C++ Builderu vlastně kreslí.

Canvas neboli plátno

Ke kreslení, psaní textu apod. slouží v C++ Builderu komponenta Canvas. Není na paletách, ale některé další komponenty — např. okno (TForm), bitová mapa (TBitmap) nebo obrázek (komponenta TImage) — ji obsahují jako vlastnost (property).

Komponenta Canvas (v překladu to znamená plátno) představuje kreslicí plochu. S tímto plátnem zacházíme vždy stejně, bez ohledu na to, zda kreslíme na plochu okna nebo komponenty Image, do bitmapového obrázku uloženého v paměti nebo třeba na kreslicí plochu tiskárny. Nezáleží ani na nastaveném rozlišení grafického adaptéru našeho počítače[15]. Všechny tyto skutečnosti samozřejmě při kreslení hrají roli, my se o ně ale nemusíme starat — to je věc implementace třídy TCanvas a Windows. (Můžeme tedy tvrdit, že cavnas představuje abstraktní kreslicí prostor nezávislý na hardwarových detailech.)

Mezi vlastnostmi plátna, tedy komponenty Canvas, najdeme pero (Pen), štětec (Brush) písmo (Font) a pole grafických bodů, pixelů, které kreslicí plochu tvoří (Pixels). Všechny tyto vlastnosti jsou opět komponenty, takže mohou mít opět své vlastnosti.

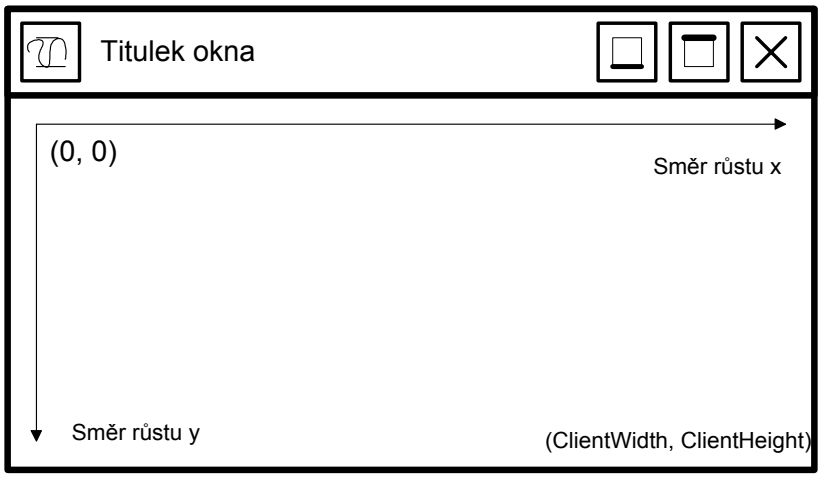
- ✧ Vlastnost Pen určuje tloušťku, barvu a styl kreslených čar.
- ✧ Vlastnost Brush určuje výplň uzavřených obrazců. Jestliže nakreslíme elipsu, vyplní se její vnitřek daným „štětcem“. Můžeme si předepsat vyplnění souvislou barevnou plochou, daným vzorem nebo danou bitovou mapou.
- ✧ Vlastnost Font určuje typ písma, jeho velikost, barvu atd.
- ✧ Vlastnost Pixels umožňuje pracovat s jednotlivými body obrázku — zjišťovat nebo nastavovat jejich barvu. Je implementována jako dvourozměrné pole barevných bodů.

Vedle toho nám plátno dává k disposici řadu metod pro kreslení obrazců, pro zobrazení hotové bitové mapy, změnu měřítka atd. Následující přehled ukazuje jen některé z nich, podrobnější informace najdete v nápovědě.

Arc()	Nakreslí aktuálním perem oblouk elipsy.
Draw()	Vykreslí bitovou mapu, ikonu nebo metasoubor.

[15] Pokud znáte aplikační rozhraní Windows, postačí, když řekneme, že TCanvas je zapouzdřený kontext grafického zařízení (device context).

Ellipse()	Nakreslí aktuálním perem celou elipsu.
FillRect()	Vyplní zadanou obdélníkovou oblast aktuálním štětcem.
MoveTo()	Přemístí aktuální kreslicí pozici do daného bodu.
LineTo()	Nakreslí aktuálním perem čáru z aktuální pozice do daného bodu.
Rectangle()	Nakreslí obdélník.
PolyBezier()	Nakreslí sadu Bézierových křivek.
StretchDraw()	Podobná jako Draw(), změní však rozměry obrázku tak, aby vyplnil danou plochu.
TextOut()	Vypíše daný text od aktuálního kreslicího bodu. Přitom použije nastavený font.
TextHeight(), TextWidth(), TextExtent()	Zjistí výšku, šířku resp. oba rozměry daného textu v pixelech.
Lock(), Unlock()	Uzamkne, resp. odemkne plátno. (Slouží k synchronizaci přístupu k plátnu v multithreadovém programu.)
Refresh()	Vrátí kontext zařízení, který plátno používá, do původního stravu.



Obr. 3.4 Implicitní souřadnicový systém v okně

Souřadnice v okně

Při kreslení na plátno, canvas, vlastně buď přímo nebo prostřednictvím různých funkcí měníme barvu jednotlivých bodů. Proto musíme umět tyto body zadat.

V oknech se vzdálenost implicitně měří v pixelech (obrazových bodech). Počátek soustavy souřadnic, tj. bod (0, 0), je v levém horním rohu klientské oblasti okna. Souřadnice *x* roste ve směru zprava doleva, souřadnice *y* ve směru shora dolů; viz obr. 3.4. Pozor, směr, ve kterém roste *y*, může z počátku působit problémy.

Souřadnice bodu v pravém dolním rohu okna jsou (ClientWidth, ClientHeight), kde ClientWidth a ClientHeight jsou vlastnosti, které určují rozměry klientské části okna.

Začínáme

Začneme nový projekt, který uložíme pod jménem hodiny3, a modul s oknem uložíme pod jménem cifernik.cpp. Vymyslíme vhodný nápis do titulkové lišty, rozměry okna a barvy pozadí okna. Při kreslení hodin jde vlastně o tři věci: Nakreslení ciferníku, zjištění času a jeho rozložení na hodiny, minuty a sekundy a nakonec o nakreslení ručiček.

Ciferník

Nejprve určíme střed klientské oblasti okna, neboť to bude i střed ciferníku, ve kterém budou začínat ručičky. K tomu použijeme vlastnosti ClientWidth, resp. ClientHeight okna cifernik; tyto vlastnosti obsahují aktuální šířku, resp. výšku klientské oblasti okna (oblasti, do které program běžně kreslí — nezahrnuje titulkovou lištu a okraj). Souřadnice středu jsou

$x_s = ClientWidth / 2, \quad y_s = ClientHeight / 2.$

Ciferník bude kruh o poloměru o něco menším než je polovina menšího z rozměrů klientské oblasti, např.

$R = \min (x_s, y_s) - 10,$

Ciferník zpravidla obsahuje čísla označující hodiny. My se spokojíme s dvanácti kruhovými značkami o poloměru rovném např. $^1/_{40}$ poloměru ciferníku, nejméně však 1 bod. Tyto značky budou rozmístěny po obvodu kruhu v pravidelných intervalech. Jako první nakreslíme značku v místě, kde bývá číslice 3; tato značka bude mít souřadnice $(x_s + R, y_s)$.

Další značky následují po 30°, tedy po $^\pi/_6$. Svírá-li spojnice značky se středem ciferníku úhel φ, má značka na obrazovce souřadnice

$x = x_s + R * \cos \varphi, \quad y = y_s - R * \sin \varphi.$

Znaménko minus ve druhém vzorci je dáno tím, že souřadnice y roste shora dolů.

K nakreslení značek použijeme metodu Canvas -> Ellipse(). Tato funkce má jako parametry souřadnice levého horního a pravého dolního rohu obdélníka, do kterého je vepsána. Při kreslení použijeme stejnou barvu pera (tedy čáry) a štětce (tedy výplně), takže dostaneme jednobarevný kruh.

Čas

Aktuální systémový čas zjistíme podobně jako v předchozím oddílu voláním funkce Time(). Vrácenou hodnotu potřebujeme rozložit na jednotlivé údaje. K tomu nám poslouží funkce DecodeTime(), která uloží výsledek (hodiny, minuty, sekundy a milisekundy) do čtyř proměnných typu unsigned short.

Ručičky

Nyní zbývá už jen nakreslit ručičky tak, aby ukazovaly aktuální čas. Ručičky zobrazíme jako čáry vedoucí ze středu ciferníku. Jednotlivé ručičky odlišíme barvou, délkou a tloušťkou čáry. Hodinová ručička bude mít délku $^3/_4$ poloměru ciferníku a nakreslíme ji perem o síle 3 body. Minutová ručička bude mít délku rovnou poloměru ciferníku, stejnou barvu jako hodinová ručička a tloušťku 2 body. Vteřinová ručička bude mít stejnou délku jako minutová, tloušťku 1 bod a jinou barvu.

Hodinová ručička proběhne za 12 hodin celý kruh, tedy 2π. To znamená, že za 1 hodinu uběhne $^\pi/_6$, takže změna úhlu hodinové ručičku za hodinu bude $\Delta_h = {}^\pi/_6$. Podobně určíme, že změna úhlu minutové ručičky za 1 minutu bude $\Delta_m = {}^\pi/_{30}$. Stejná bude i změna úhlu vteřinové ručičky za 1 vteřinu.

V čase 0:0:0 jsou všechny tři ručičky svislé, směřují od středu ciferníku vzhůru. Poloha hodinové a minutové ručičky se bude měnit každou minutu, poloha vteřinové pochopitelně každou vteřinu. V čase h:m:s proto bude svírat hodinová, minutová a vteřinová ručička se svislým směrem úhel

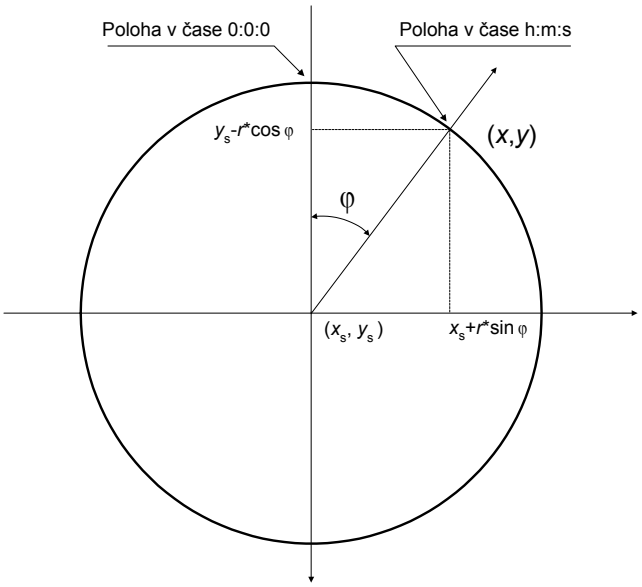
$\varphi_h = (h + m/60) * \Delta_h, \quad \varphi_m = m * \Delta_m, \quad \varphi_s = s * \Delta_m.$

takže souřadnice koncových bodů minutové a vteřinové ručičky budou

$x = x_s + R * \sin \varphi, \quad y = y_s - R * \cos \varphi,$

kde φ je podle okolností φ_m nebo φ_s, a souřadnice koncového bodu hodinové ručičky budou (viz též obr. 3.5)

$x = x_s + 0{,}75 * R * \sin \varphi_h, \quad y = y_s - 0{,}75 * R * \cos \varphi_h.$



Obr. 3.5 Souřadnice konce ručiček

Nyní můžeme napsat funkci[16] NakresliHodiny():

```
void Tcifernik1::NakresliHodiny()
{
 int XS =ClientWidth /2, YS = ClientHeight / 2;       // Souřadnice středu
 int R = min (XS, YS)-10;                             // Poloměr ciferníku
 const double deltah = M_PI / 6;                      // Přírůstek úhlu za hodinu
 const double deltam = M_PI / 30;                     // Přírůstek úhlu za minutu

 // Kreslení značek na ciferník
 Canvas -> Pen -> Color = clYellow;                   // Pero a štětec pro značky
 Canvas -> Brush -> Color = clYellow;
 int r = R/40.0+1;                                    // Poloměr značky
 double fi = 0;                                       // Počáteční úhel

 for(int i = 0; i < 12; i++, fi += deltah)            // Značky hodin
 {
  int x = XS + R * cos(fi);
  int y = YS + R * sin(fi);
  Canvas -> Ellipse(x-r, y-r, x+r, y+r);
 }

 unsigned short h, m, s, ms;                          // Zjištění a dekódování času
 DecodeTime(Time(), h, m, s, ms);

 Canvas -> Pen -> Width = 4;                          // Barva a síla pera pro hodinovou
 Canvas -> Pen -> Color = clNavy;                     // a vteřinovou ručičku

 // Kreslení hodinové ručičky
 fi = deltah * (h + m/60.0);                          // Úhel ručičku se svislicí
 Canvas -> MoveTo(XS, YS);
 Canvas -> LineTo(XS + 0.75*R*sin(fi), YS - 0.75*R*cos(fi));

 // Kreslení minutové ručičky
 Canvas -> Pen -> Width = 2;
 fi = deltam * m;
 Canvas -> MoveTo(XS, YS);
 Canvas -> LineTo(XS + R*sin(fi), YS - R*cos(fi));

 // Kreslení vteřinové ručičky
 Canvas -> Pen -> Width = 1;
 Canvas -> Pen -> Color = clRed;
 fi = deltam * s;
 Canvas -> MoveTo(XS, YS);
 Canvas -> LineTo(XS + R*sin(fi), YS - R*cos(fi));
}
```

[16] M_PI je předdefinovaná konstanta (makro) s hodnotou π. Najdeme ji v hlavičkovém souboru math.h. Nejde o standardní součást C++, ale o borlandské rozšíření.

Výpis funkce NakresliHodiny()

Teď si musíme ujasnit, jak bude náš program fungovat. Pokud bychom funkci NakresliHodiny() prostě volali vždy při události OnTimer, skládaly by se jednotlivé obrázky přes sebe, a to nechceme. My potřebujeme vždy starý obrázek vymazat a nakreslit nový s novou polohou ručiček. K tomu nám poslouží metoda Repaint(), kterou náš formulář zdědil po předkovi. Tato metoda vymaže obsah okna a způsobí jeho překreslení (vyvolá událost OnPaint).

Vedle toho ale musíme hodiny znovu nakreslit vždy poté, co jejich okno překryje a zase uvolní okno jiného programu nebo když změníme velikost okna[17]. To znamená, že musíme reagovat na události OnTimer, OnPaint a OnResize. Jejich handlery budou velice jednoduché: Při událostech OnTimer a OnResize prostě zavoláme metodu Repaint(), při události OnPaint zavoláme funkci NakresliHodiny().

```
void __fastcall Tcifernik1::nepokojTimer(TObject *Sender)
{
  Repaint();
}
void __fastcall Tcifernik1::FormResize(TObject *Sender)
{
  Repaint();
}
void __fastcall Tcifernik1::FormPaint(TObject *Sender)
{
  NakresliHodiny();
}
```

Ošetření událostí OnTimer, OnPaint a OnResize

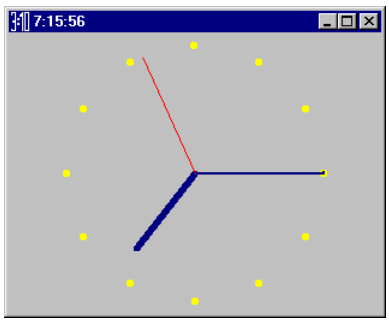
Zbývá ještě připojit k programu ikonu, přeložit ho a spustit.

Co dál

Náš program opět není příliš dokonalý, a tak doporučuji zkusit ho vylepšit. Můžete zkusit odstranit titulkovou lištu, podobně jako u našeho předchozího programu. Můžete titulkovou lištu ponechat a napsat do ní čas, takže vznikne kombinace analogových a digitálních hodin. K tomu stačí napsat na konec funkce NakresliHodiny() příkaz

```
Caption = TimeToStr(Time());
```

(viz obr. 3.6). Program v této podobě najdete na CD v adresáři KAP03\03.



Obr. 3.6 Hodiny s ručičkami

Lepší verze

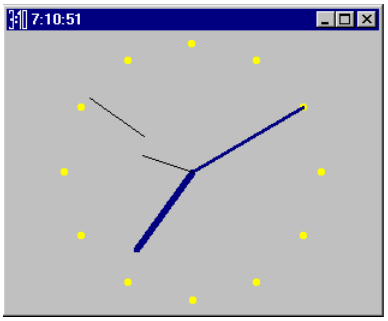
Naše hodiny sice jdou, ale příliš kvalitní program to stále není. Jeho zjevnou vadou je neustálé poblikávání.

Druhá chyba našeho programu se zatím nemá možnost příliš projevit, neboť interval překreslování, jedna sekunda, je příliš krátký, než abychom si jí mohli všimnout. Udělejme proto

[17] Jedním ze základních principů programování pro Windows je, že program se musí starat obsah svého okna. Jestliže operační systém usoudí, že je třeba obsah určitého okna překreslit, pošle programu zprávu WM_PAINT; ta v programu napsaném v C++ Builderu způsobí vznik události OnPaint. Poznamenejme, že komponenty TLabel a TPanel, které jsme dosud používali, se o aktualizaci svého výstupu starají automaticky.

následující pokus: Prodlužme interval časovače např. na 10 sekund (nastavíme vlastnost Interval na 10000), spusťme program, zakryjme část okna jakýmkoli jiným oknem a zase ji odkryjme. Nejspíš pak na chvilku uvidíme něco podobného jako na obr. 3.7. Při nejbližším přesunu vteřinové ručičky se obrázek zase srovná, ale hezké to není.

Problém vznikl proto, že náš program překreslil pouze část okna, kterou jsme předtím zakryli, která je „neplatná“. To není libovůle C++ Builderu, ale standardní chování programů pod Windows: Při ošetřování zprávy WM_PAINT (a tedy v Builderu při reakci na událost OnPaint) se překreslí pouze „neplatná oblast“, tj. nejmenší obdélník, do kterého lze uzavřít poškozenou část okna.



Obr. 3.7 Lámat ručičky — to si program smí dovolit??

Metoda Repaint() označí celou klientskou oblast okna za neplatnou, takže způsobí překreslení celého obrázku. Mohlo by se tedy zdát rozumné zavolat v metodě NakresliHodiny() nebo v handleru FormPaint() nejprve metodu Repaint(). Výsledkem by ale byl nepřetržitě blikající program, neboť metoda Repaint() označí obsah okna za neplatný, takže ihned po dokončení se začne kreslit znovu (znovu nastane událost OnPaint), a to se bude opakovat v nekonečném cyklu.

Řešení je trochu jiné: Oddělíme od sebe kreslení a získávání údaje o čase.

Handler, který ošetřuje událost OnTimer, nejprve zjistí čas, dekóduje ho a získaná data uloží do vhodných datových složek okna (třídy Tcifernik). Teprve pak požádá o překreslení.

Handler, který ošetřuje událost OnPaint, se bude řídit uloženými daty. To znamená, že pokud dojde k překreslování z jiného důvodu než kvůli změně velikosti okna nebo kvůli jeho zakrytí, použije se čas, podle kterého byl nakreslen poškozený obrázek, takže se nakreslí se stejný obrázek jako předtím.

Tím odstraníme lámání ručiček; zbývá ještě odstranit blikání způsobené tím, že každou sekundu překreslujeme celý obsah okna. To je ale zbytečné, stačilo by smazat vždy ručičky a nakreslit je v nové poloze. I zde se nabízí jednoduché řešení: Při kreslení ručiček použijeme pero v režimu pmXor. V tomto režimu se vždy dělá bitová nonekvivalence kreslených bodů s body pozadí, a protože pro libovolné a, b platí (a^b)^b = a, stačí ručičku nakreslit podruhé na stejné místo a tím ji smažeme.

Nové schéma programu tedy bude následující:

- ✧ Data, která popisují ciferník (souřadnice středu, poloměr atd.) a poslední zjištěný časový údaj uložíme do vhodných datových složek třídy Tcifernik1.
- ✧ Metodu pro kreslení hodin rozdělíme do tří samostatných procedur: Znacky(), která nakreslí značky ciferníku, PomaleRucicky(), která nakreslí hodinovou a minutovou ručičku, a Vterinovka(), která nakreslí vteřinovou ručičku. Procedury pro kreslení ručiček budou mít jako parametry čas, který mají zobrazit.
- ✧ Handler ošetřující událost OnPaint prostě nakreslí ciferník a ručičky. Použije k tomu uložený časový údaj.
- ✧ Handler ošetřující událost OnResize zjistí rozměry okna, z nich odvodí parametry ciferníku a uloží je do odpovídajících datových složek. Pak si vyžádá překreslení okna.
- ✧ Handler ošetřující událost OnTimer nejprve zjistí aktuální čas. Pak smaže vteřinovou ručičku a nakreslí ji v nové poloze. Dále porovná starý a nový údaj o počtu minut a pokud došlo ke změně, překreslí i „pomalé“ ručičky. Nakonec uloží nové časové údaje.

Podstatné části programu ukazují následující výpisy:

```
class Tcifernik1 : public TForm
{
__published:                            // IDE-managed Components
  TTimer *nepokoj;void __fastcall FormResize(TObject *Sender);
  void __fastcall nepokojTimer(TObject *Sender);
```

```
  void __fastcall FormPaint(TObject *Sender);
private:                                        // Data potřebná pro kreslení hodin:
  int XS, YS;                                   // Souřadnice středu okna
  double deltah, deltam;        // Přírůstek úhlu ručičky za hodinu, minutu
  int R, r;                                     // Poloměr ciferníku a značky
  unsigned short h, m, s;  // Čas
   void Znacky();                       // Zna ky na cifern ku
  void PomaleRucicky(unsigned short hod, unsigned short min);
  void Vterinovka(unsigned short sec);
public:
  __fastcall Tcifernik1(TComponent* Owner);
};
```

Definice třídy Tcifernik1 v souboru cifernik.h

```
Tcifernik1 *cifernik1;
__fastcall Tcifernik1::Tcifernik1(TComponent* Owner) // Konstruktor vypo te
  : TForm(Owner), deltah(M_PI/6), deltam(M_PI/30)// konstanty, zjistí
{                                                                                       // čas v okamžiku spuštění
  unsigned short ms;
  DecodeTime(Time(), h, m, s, ms);                      // a uloží ho
}
inline int min(int a, int b) { return a < b ? a : b; }
void Tcifernik1::Znacky()               // Nakresl  zna ky na cifern ku
{
 Canvas -> Pen -> Color = clYellow;
 Canvas -> Brush -> Color = clYellow;
  double fi = 0;                                // Po   te n   hel
 for(int i = 0; i < 12; i++)            // Značky hodin
 {
  int x = XS + R * cos(fi);
  int y = YS + R * sin(fi);
  Canvas -> Ellipse(x-r, y-r, x+r, y+r);
  fi += deltah;
 }
}
void Tcifernik1::PomaleRucicky(unsigned short hod, unsigned short min)
{
 Canvas -> Pen -> Width = 4;
 Canvas -> Pen -> Color = clYellow;
 TPenMode pmStare =  Canvas -> Pen -> Mode;    // Uložíme původní režim
 Canvas -> Pen -> Mode = pmXor;                 // Nastavíme pmXor
  // Hodinov  ru ka
 double fi = deltah * (hod + min/60.0);         // Vypočteme úhel
 Canvas -> MoveTo(XS, YS);                      // Nakreslíme ručičku
 Canvas -> LineTo(XS + 0.75*R*sin(fi), YS - 0.75*R*cos(fi));
  // Minutov  ru ka
 Canvas -> Pen -> Width = 2;
 fi = deltam * min;                             // Vypočteme úhel
 Canvas -> MoveTo(XS, YS);                      // Nakreslíme ručičku
 Canvas -> LineTo(XS + R*sin(fi), YS - R*cos(fi));
 Canvas -> Pen -> Mode = pmStare;               // Vrátíme původní režim
}
void Tcifernik1::Vterinovka(unsigned short sec)
{
 // Vteřinová ručka
 Canvas -> Pen -> Width = 1;
 Canvas -> Pen -> Color = clWhite;
 TPenMode pmStare =  Canvas -> Pen -> Mode;    // Uložíme původní režim
 Canvas -> Pen -> Mode = pmXor;                 // Nastavíme pmXor
 double fi = deltam * sec;                      // Vypočteme úhel
 Canvas -> MoveTo(XS, YS);                      // Nakreslíme ručičku
 Canvas -> LineTo(XS + R*sin(fi), YS - R*cos(fi));
 Canvas -> Pen -> Mode = pmStare;               // Vrátíme původní režim
}
void __fastcall Tcifernik1::FormResize(TObject *Sender)
{
  // Střed a poloměr ciferníku, poloměr značek
  XS =ClientWidth /2;                           // Zjistíme údaje o ciferníku
  YS = ClientHeight / 2;
  R = min (XS, YS)-10;
```

```
  r = R/50+1;
  Repaint();                                    // a vyžádáme si překreslení
}
void __fastcall Tcifernik1::nepokojTimer(TObject *Sender)
{
  unsigned short ms;                            // Pomocná proměnná
  unsigned short Hnove, Mnove, Snove;           // zjistíme nový čas
  DecodeTime(Time(), Hnove, Mnove, Snove, ms);
  Vterinovka(s);                                // Smažeme vteřinovku
  Vterinovka(Snove);                            // a nakreslíme ji znovu vedle
  s = Snove;                                    // Uložíme nový čas
  if(m != Mnove)                                // Pokud se změnily minuty
  {
    PomaleRucicky(h, m);                        // smažeme i další ručičky
    PomaleRucicky(Hnove, Mnove);                // a nakreslíme je znovu vedle
    h = Hnove; m = Mnove;                       // Uložíme nový čas
  }
}
void __fastcall Tcifernik1::FormPaint(TObject *Sender)
{
  Znacky();                                     // Nakreslíme značky
  PomaleRucicky(h, m);                          // a ručičky
  Vterinovka(s);
  Caption = AnsiString(h)+":"+AnsiString(m)+":"+s;
}
```

Výpis souboru cifernik.cpp

Poznámky

- ✧ V režimu pmXor může být problém předem odhadnout barevný výsledek. Zde jsme např. zadali žlutou barvu pro „pomalé“ ručičky a bílou pro vteřinovou, nicméně výsledek je — při šedém pozadí — černá vteřinová ručička a modré zbývající dvě.
- ✧ Náš program také vypisuje čas do titulkové lišty. Vlastnost Caption očekává znakový řetězec uložený v typu AnsiString (třída pro práci se znakovými řetězci, která *neodpovídá* standardu ANSI jazyka C++). Ve třídě AnsiString jsou definovány konstruktory, které umožňují převod číselných hodnot, typu char* a dalších na instance této třídy. Dále je pro třídu AnsiString definován operátor +, který umožňuje spojování instancí.
- ✧ Všimněte si, že nyní se handler události OnPaint volá jako výsledek změny velikosti okna nebo jeho „poškození“. Volá se také při vytvoření okna na počátku programu, nikoli však při zpracování zprávy od časovače — o kreslení při změně časového údaje se stará handler ošetřující časovač.
- ✧ Upravený program najdete na CD v adresáři KAP03\04.

Hodinky s vodotryskem

Myslím, že nadešel čas na obligátní otázku: Co byste ještě chtěli? Hodinky s vodotryskem?

Dobře, uděláme si je.

Vyjdeme z předchozí verze programu; připojíme k ní nové okno, které bude cyklicky zobrazovat několik bitmapových obrázků. Tyto obrázky vytvoříme např. v editoru obrázků a k programu je připojíme jako tzv. prostředky (resources). Střídání obrázků bude řídit další časovač. Vodotrysk vyvoláme stisknutím kombinace ALT+PRAVÉ TLAČÍTKO MYŠI a poběží jen krátkou dobu, pak zase zmizí.

Prostředky (vodotrysk)

Nejprve si pomocí editoru obrázků vytvoříme tři bitové mapy o rozměrech 100 × 100 pixelů, které budou zobrazovat různé, i když velmi blízké tvary vodotrysku, a uložíme je do adresáře projektu pod názvy Voda1.bmp, Voda2.bmp a Voda3.bmp.

Pak vytvoříme soubor s popisem prostředků. Příkazem File | New... vyvoláme zásobník objektů a na kartě New vybereme položku Text. Vytvoří se prázdný textový soubor, do kterého napíšeme následující popis našich tří bitových map.

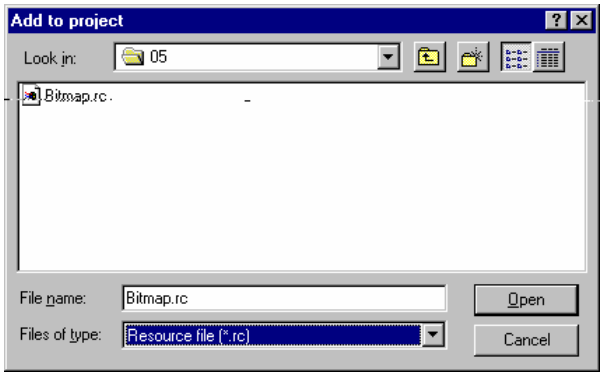
```
BMP1 BITMAP Voda1.bmp
```

```
BMP2 BITMAP Voda2.bmp
BMP3 BITMAP Voda3.bmp
```

Soubor Bitmap.rc

Formát tohoto popisu je jednoduchý: Popis začíná identifikátorem, pod kterým budeme bitovou mapu v programu používat (zde BMP1 až BMP3). Pak následuje klíčové slovo BITMAP a jméno souboru, ve kterém požadovaná bitová mapa je.

Vytvořený soubor uložíme jako soubor s prostředky (resource file) pod názvem Bitmap.rc. Pak jej připojíme k našemu projektu. Příkazem Project | Add to Project... vyvoláme dialogové okno Add to project (obr. 3.8) a v něm určíme, že chceme připojit soubor s prostředky s daným jménem. Pak stiskneme tlačítko Open.



Obr. 3.8 Připojujeme soubor k projektu

Soubory s prostředky převede program zvaný překladač prostředků, resource compiler, do binární podoby[18] (soubor .RES). Při sestavování se pak připojí jako data ke spustitelnému souboru (.EXE). V programu naše bitové mapy použijeme pomocí metody LoadFromResourceName(); o tom si povíme dále v oddílu Animace.

Okno s obrázkem

Dále k našemu projektu připojíme nové okno, které uložíme pod názvem vodotrysk do souboru voda.cpp. Protože jde o druhé okno aplikace, bude jeho vlastnost Visible implicitně false. Do tohoto okna vložíme časovač, který bude řídit animaci; jeho vlastnost Interval nastavíme na 300 ms. Vlastnost Enabled nastavíme na false, neboť pokud okno s vodotryskem není vidět, je zbytečné, aby tento časovač běžel. Handler, který bude reagovat na zprávy od časovače, napíšeme později.

Rozměry okna upravíme tak, aby jeho klientská oblast měla rozměry přibližně 100 × 100 bodů, a okno umístíme nad ciferník přibližně doprostřed. Přesné umístění uděláme až za běhu programu. Pak odstraníme titulkovou lištu okna. To už umíme — stačí předefinovat metodu CreateParams tak, aby definovala okno se stylem WS_POPUP:

```
void __fastcall Tvodotrysk::CreateParams(TCreateParams &Par)
{
  TForm::CreateParams(Par);
  Par.Style = WS_POPUP| WS_THICKFRAME;
}
```

Do okna vložíme komponentu Image, kterou najdeme na paletě Additional. Pojmenujeme ji obrazek, neboť bude zobrazovat obrázky s vodotryskem. Její vlastnost Align nastavíme na hodnotu alClient, takže vyplní celou klientskou oblast okna, a vlastnosti AutoSize přidělíme hodnotu true.

Vyvolání okna: reakce na myš

Okno s vodotryskem chceme vyvolat stisknutím kombinace ALT+PRAVÉ TLAČÍTKO MYŠI na ciferníku hodin. To znamená, že okno s ciferníkem musí reagovat na stisknutí tlačítka myši; napíšeme tedy handler pro událost OnMouseDown. Tento handler má prototyp

```
void __fastcall Tcifernik1::FormMouseDown(TObject *Sender,
                    TMouseButton Button, TShiftState Shift, int X, int Y);
```

[18] Jinou možností bylo uložit vytvořené bitové mapy přímo z editoru obrázků do souboru .RES nebo .DCR

První parametr, Sender neboli odesílatel, obsahuje jako vždy adresu objektu, který tuto událost vyvolal. Druhý parametr, Button, určuje, které tlačítko bylo stisknuto. Může nabývat některé z hodnot mbLeft, mbRight nebo mbMiddle (levé, pravé nebo prostřední tlačítko). Parametr Shift určuje, které z přeřaďovačů byly v době vzniku události stisknuty. Je typu TShiftState, což je analogie pascalské množiny. Jednotlivé přeřaďovače jsou popsány příznaky ssAlt, ssShift, ssCtrl. (Další příznaky popisují levé tlačítko myši, dvojklik atd. — podrobnosti najdete v nápovědě.) Chceme-li zjistit, zda tato množina obsahuje určitý příznak, použijeme metodu TShiftState::Contains().

Poslední dva parametry handleru, X a Y, obsahují souřadnice bodu v klientské oblasti odesílatele, ve které se nacházel kurzor myši v okamžiku, kdy k události došlo.

Handler nejprve zjistí, zda jsme stiskli pravé tlačítko myši zároveň s klávesou ALT. Pokud ano, upraví velikost okna vodotrysk, přemístí ho doprostřed nad ciferník, zobrazí ho a spustí časovač. Úplný kód této funkce ukazuje následující výpis.

```
void __fastcall Tcifernik1::FormMouseDown(TObject *Sender,
    TMouseButton Button, TShiftState Shift, int X, int Y)
{               // Je to pravé tlačítko myši + klávesa Alt?
  if(Button == mbRight && Shift.Contains(ssAlt))
  {
    vodotrysk -> ClientHeight = 100;                    // Nastav rozměry okna
    vodotrysk -> ClientWidth = 100;
                //        Umísti okno doprostřed nad ciferník
    vodotrysk->Left = cifernik1->Left + (cifernik1->Width - vodotrysk->Width)/2;
    vodotrysk->Top = cifernik1->Top - vodotrysk->Height;
    vodotrysk -> Visible = true;                        // Zobraz okno
    vodotrysk -> animator -> Enabled = true;            // Spusť časovač
  }
}
```

Handler ošetřující událost OnMouseDown

Animace

Nyní se konečně dostáváme k vlastní animaci. O ni se postará handler ošetřující událost OnTimer vyvolanou časovačem animator (v okně vodotrysk). Úkol tohoto handleru je poměrně jednoduchý:

- ✧ Nejprve nastaví správnou polohu okna s vodotryskem, neboť od posledního zobrazení ji mohl uživatel změnit.
- ✧ Pak přečte obrázek, který je na řadě, a zobrazí ho v komponentě obrazek typu TImage. Komponenta TImage má vlastnost Picture, která má vlastnost Bitmap. Chceme-li přečíst obrázek uložený jako prostředek, použijeme metodu Bitmap::LoadFromResourceName().
- ✧ Zvýší hodnotu počítadla obrázků i o 1.
- ✧ Jestliže už proběhl předepsaný počet cyklů, skryje okno, vypne časovač a vynuluje počítadlo obrázků, abychom mohli animaci příště spustit znovu.

Kód tohoto handleru a třídy Tvodotrysk ukazují následující výpisy.

```
void __fastcall Tvodotrysk::animatorTimer(TObject *Sender)
{
  static AnsiString co[3] = {"BMP1", "BMP2", "BMP3"};  // Jména obrázků
  static int i = 0;                                    // Počítadlo obrázků

            // P esu  okno na spr vné m sto
  Left = cifernik1->Left + (cifernik1->Width - Width)/2;
  Top = cifernik1->Top - Height;

            // Zobraz n sleduj c  obr zek
  obrazek -> Picture -> Bitmap ->
                          LoadFromResourceName(int(HInstance), co[i++%3]);
  if(i > kolik)
  {           // Pokud se vyčerpal předepsaný počet zobrazení, vypni časovač
    vodotrysk -> animator -> Enabled = false;
    vodotrysk -> Visible = false;   // Zavři okno s vodotryskem a uveď
    i = 0;                                             // počítadlo obrázků do počátečního stavu
  }
}
```

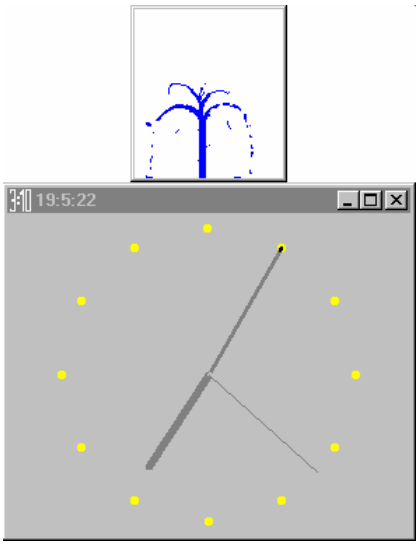
Vlastní animace (handler ošetřující událost OnTimer, soubor voda.cpp)

```
class Tvodotrysk : public TForm
{
```

```
__published:  // IDE-managed Components
   TTimer *animator;
   TImage *obrazek;
   void __fastcall animatorTimer(TObject *Sender);
private:
   static const int kolik = 12;   // Délka animace (počet zobrazených obrázků)
public:
   __fastcall Tvodotrysk(TComponent* Owner);
   void __fastcall CreateParams(TCreateParams &Par);
};
```

Výpis třídy Tvodotrysk v souboru voda.h

Po spuštění programu stiskneme klávesu ALT a klikneme pravým tlačítkem myši uprostřed ciferníku. Měli bychom na chvilku uvidět něco podobného jako na obr. 3.9.



Obr. 3.9 Hodinky s vodotryskem

Poznámky

- Prvním parametrem metody LoadFromResourceName() je identifikační číslo[19] (handle) instance programu. Najdeme je v globální proměnné HInstance. Z jakýchsi nejasných důvodů je ale musíme přetypovat na int. Druhým parametrem této funkce je znakový řetězec, který identifikuje požadovanou bitovou mapu. V našem případě to bude jedem z řetězců "BMP1", "BMP3", "BMP3", které jsme použili v souboru bitmap.rc. Tyto řetězce jsme si uložili do pole co[3] typu AnsiString.
- Prostředky v souborech .RC bychom mohli popsat také celými čísly. Pak bychom ke čtení použili funkci LoadFromResourceID(), která má jako druhý parametr odpovídající číslo.
- Lokální proměnná i slouží jako počítadlo obrázků. Musí být statická, aby se její hodnota uchovala mezi jednotlivými voláními. (Nezapomeňte, že obrázky se střídají při události OnTimer, tedy při volání metody Tvodotrysk::animatorTimer().) Proto se také musí po skončení animace vynulovat. Hodnota proměnné i roste od 0 do kolik-1, kde kolik je konstanta definovaná ve třídě Tvodotrysk.
- Deklarace static const int kolik = 12; v souboru voda.cpp není chybná; nový standard jazyka C++ umožňuje inicializovat konstantní statické složky přímo v popisu třídy.
- Uložení bitových map jako prostředků nepředstavuje jedinou možnost. Obrázky s vodotryskem můžeme také ponechat v samostatných souborech a číst je pomocí metody TBitmap::LoadFromFile().
- Prostředky, jinak také zvané *zdroje*, obsahují v tradičních programech pro Windows popisy ikon, kurzorů, bitových map, nabídek, dialogových oken, tabulek znakových řetězců atd. a jsou pravidelnou součástí projektů. V C++ Builderu je používáme daleko méně, neboť většina z těchto nástrojů je zapouzdřena do vizuálních komponent.

[19] Ve starších verzích Windows představoval typ HANDLE opravdu celé číslo bez znaménka (UINT, tedy unsigned int). V současné verzi ovšem znamená ukazatel typu void*, takže označení „identifikační číslo“ je poněkud nepřípadné. Nenapadá mne ale lepší český ekvivalent, a tak ho stále používám.

- ✧ C++ Builder obsahuje od verze 3 také komponentu Animate (na paletě Win32). Ta se však pro naše účely nehodí, neboť může přehrávat pouze soubory .AVI (a to se značnými omezeními, mj. bez zvuku) nebo tradiční animace z Win32, které se používají ve standardních dialogových oknech při vyhledávání souborů apod.
- ✧ Hodinky s vodotryskem najdete na CD v adresáři KAP03\05.

3.3 Hodinky v pruhu úloh

Na konec jsme se ponechali nejužitečnější možnost: Hodinky, které nepotřebují okno, takže nepřekážejí na pracovní ploše, a čas zobrazují v nápisu na tlačítku v pruhu úloh (taskbar).

Nápis na tlačítku je určen vlastností Title objektu Application.

Vytvoříme tedy nový projekt, který uložíme pod jménem hodiny6. Soubor s oknem uložíme pod názvem okno, formulář pojmenujeme vtipně okno1. Vlastnost formuláře WindowState nastavíme na wsMinimized, takže se okno na počátku vytvoří minimalizované. Vlastnost Visible nastavíme na false.

Do návrhu okna vložíme časovač, který nazveme casovac; jeho vlastnost Interval nastavíme na 1000. Handler, který bude ošetřovat událost OnTimer, zjistí čas, převede ho na znakový řetězec a přiřadí ho vlastnosti Application -> Title. Přidělíme programu ikonu, přeložíme ho a spustíme. Výsledek ukazuje obr. 3.10.

Obr. 3.10 Další hodiny v pruhu úloh Windows

Náš program má jednu vadu na kráse: Po spuštění se na tlačítku v pruhu úloh nejprve objeví nápis hodiny6 a teprve asi po 1 vteřině čas. Ale i to odstraníme jednoduše. Při spuštění programu, tj. při vytvoření okna, přiřadíme vlastnosti Application -> Title prázdný řetězec. K tomu použijeme handler ošetřující událost OnCreate, která nastává při vytvoření okna.

Zdrojový text našeho programu je tentokrát opravdu jednoduchý:

```
// Konstruktor t   dy okna
__fastcall Tokno1::Tokno1(TComponent* Owner)
  : TForm(Owner)
{}

//   asova
void __fastcall Tokno1::casovacTimer(TObject *Sender)
{
  Application -> Title = TimeToStr(Time());  // Vypiš čas
}

// P i vytvo en   okna nastav me pr zdn   titulek
void __fastcall Tokno1::FormCreate(TObject *Sender)
{
  Application -> Title = "";
}
```

Výpis souboru okno.cpp

Poznámky

- ✧ Program ukončíme příkazem Close ze systémového menu, které vyvoláme stisknutím pravého tlačítka myši na tlačítku programu na pruhu úloh.
- ✧ Pokud uživatel klikne na tlačítko hodin, uvidí prázdné okno. Můžete ho zkusit vyzdobit vhodným nápisem; můžete také zkusit ho zase zavřít a pozorovat účinky různých příkazů.
- ✧ Program hodiny6 najdeme na CD v adresáři KAP03\06.
- ✧ Může se zdát, že psát hodiny je zbytečné, neboť jedny už v operačním systému jsou. No a?
- ✧ V 10. kapitole se k hodinám ještě vrátíme — vytvoříme je jako komponentu.

4. Prohlížeč obrázků

V této kapitole napíšeme jednoduchý prohlížeč bitmapových obrázků. Přitom se mj. seznámíme s některými standardními dialogy, vytvoříme nástrojový panel a naučíme se psát aplikace s rozhraním MDI.

4.1 První verze

Vytvoříme nový projekt; soubor s hlavním programem uložíme pod názvem prohl.cpp, modul popisující okno pod názvem obraz.cpp. Pak do vizuálního návrhu okna našeho programu vložíme komponentu MainMenu, kterou pojmenujeme menu, a vytvoříme nabídku s položkami Soubor a Nápověda. Odpovídající komponenty typu TMenuItem pojmenujeme výstižně soubor a napoveda.

Nabídka Soubor bude obsahovat položky Otevři..., Ulož jako..., Zavři a Konec, menu Nápověda bude obsahovat pouze položku O programu.... Odpovídající komponenty typu TMenuItem pojmenujeme otevri, ulozjako, zavri, konec a oprogramu.

Dále do vizuálního návrhu vložíme komponentu Panel, kterou pojmenujeme panel. Její vlastnosti Align dáme hodnotu alClient, takže vyplní celou klientskou oblast okna. (Všimněte si, že nahoře zůstane volný pruh pro nabídku.) Vlastnost Caption vymažeme, takže panel neponese žádný nápis.

Na panel položíme komponentu Image (je na paletě Additional). Pojmenujeme ji obrazek a její vlastnost Align nastavíme na alClient, takže vyplní celou plochu panelu.

Nyní začneme programovat reakce na jednotlivé příkazy z nabídky.

Otevření obrázku

Začneme tím, že program naučíme otevřít soubor s obrázkem, přečíst ho a zobrazit ho. Nejdřív musíme zjistit jméno souboru a cestu k němu. K tomu můžeme použít některé ze standardních dialogových oken, které najdeme v komponentách na paletě Dialogs. Pro naše účely se hodí buď OpenDialog (pro otevření libovolného souboru) nebo OpenPictureDialog (speciálně pro obrázkové soubory). Použijeme druhý z nich, neboť nabízí náhled před otevřením.

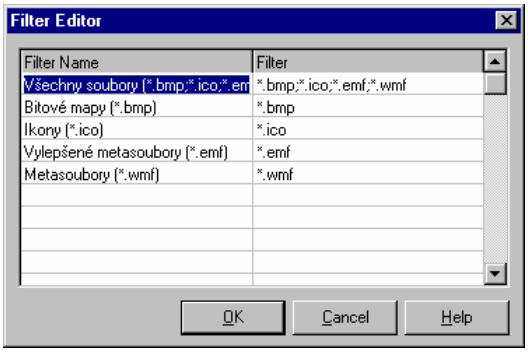
Umístíme tedy do vizuálního návrhu komponentu OpenPictureDialog a pojmenujeme ji dialog_otevri. Její vlastnost Filter určuje typy (přípony) souborů, které se v dialogovém okně budou zobrazovat; kliknutím na tlačítko se třemi tečkami si vyvoláme editor této vlastnosti (obr. 4.1), ve kterém můžeme změnit implicitní nastavení. Standardně nabízí bitové mapy (.bmp), ikony a metasoubory (.wmf a .emf). To nám vyhovuje, a proto pouze přeložíme doprovodné texty do češtiny.

Poznámka

Filtr lze zadat za běhu programu jako znakový řetězec (AnsiString) ve tvaru

"text1|přípona1|text2|přípona2..."

kde text je text, který se objeví v dialogovém okně v poli Soubory typu, a přípona je odpovídající přípona, kterou zapisujeme ve tvaru stejném jako v dialogovém okně Filter Editor.



Obr. 4.1 Editor vlastnosti Filter dialogového okna pro otevření obrázku

Vlastnost FilterIndex určuje, kterou z možností má dialogové okno nabízet jako implicitní; zadáme hodnotu 2, tedy bitové mapy. Do vlastnosti Title napíšeme Otevři obrázek — to bude titulek dialogového okna. Vlastnost Options poskytuje možnosti, jak ovlivnit vzhled a chování dialogového okna; ponecháme zatím standardní nastavení.

Jméno vybraného souboru (včetně úplné cesty a jména disku) bude uloženo ve vlastnosti FileName typu AnsiString.

Použití této komponenty je jednoduché: Všechny komponenty zapouzdřující standardní dialogová okna mají metodu Execute(), která vrátí true, pokud uživatel stiskl tlačítko Otevřít, a false, pokud stiskl tlačítko Storno. V handleru události OnClick položky Soubor | Otevři... tuto metodu zavoláme. Jakmile budeme znát jméno souboru, přečteme ho a zobrazíme v komponentě Image pomocí metody LoadFromFile() vlastnosti Picture. Náš první pokus o handler Tokno::otevriClick() by mohl vypadat takto:

```
void __fastcall Tokno::otevriClick(TObject *Sender)
{
  if(dialog_otevri -> Execute())
          obrazek->Picture -> LoadFromFile(dialog_otevri -> FileName);
}
```

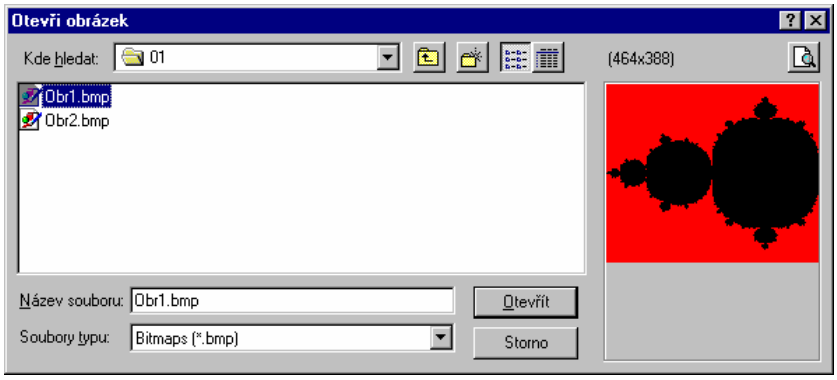
První pokus o čtení souboru s obrázkem

Jestliže nyní program přeložíme, spustíme, vybereme v nabídce položku Soubor | Otevři... a vyhledáme vhodný obrázek, dostaneme dialogové okno, které ukazuje obr. 4.2, a po stisknutí tlačítka Open se zvolený obrázek opravdu zobrazí v okně programu.[20]

Čtení souboru se nemusí podařit

Program vypadá na pohled hezky, ale jako vždy má několik zjevných i méně zjevných chyb. První z nich je, že pokud je obrázek větší, nevejde se celý na plochu okna. Trochu si můžeme vypomoci zvětšením okna, ale to není úplné řešení — existují bitové mapy větší než je plocha obrazovky. Nabízejí se dvě cesty: Zobrazit bitovou mapu na ploše, kterou máme k disposici, (tj. transformovat ji na rozměry klientské oblasti okna) nebo připojit k oknu posuvníky (scroll bar).

[20] Několik obrázků ve vhodných formátech najdete na doprovodném CD v adresáři KAP04\01. Další soubory s obrázky jsou volitelnou součástí instalace C++ Builderu a najdete je v adresáři Borland shared\Images.



Obr. 4.2 Dialogové okno pro otevření obrázku

Druhá chyba se při prvních pokusech s programem nejspíš neprojeví, ale o to je záludnější. Jestliže se funkci LoadFromFile() nepodaří grafický soubor přečíst, protože zadaný soubor neexistuje nebo je pokažený nebo protože jsme zadali soubor ve formátu, který tato funkce nezná, vyvolá výjimku typu EInvalidGraphic. Jestliže zadáme neexistující soubor, vyvolá výjimku EFopenError. S tím musíme počítat, tuto výjimku zachytit a ošetřit.

Výjimky

Uzavřeme tedy pokus o čtení do bloku try a připojíme k němu handler catch, ve kterém specifikujeme výjimky typu Exception. (Protože Exception je společným předkem všech tříd pro přenos informací o výjimkách, které se používají ve VCL, zachytíme tak úplně všechny výjimky, které zde mohou nastat.[21] Poznamenejme, že *výjimky z VCL musíme zachycovat odkazem*.)

V handleru catch vypíšeme sdělení, že se nepodařilo otevřít soubor; k tomu bychom si mohli vytvořit modální dialogové okno. C++ Builder ale nabízí lepší řešení: Použijeme metodu TApplication::MessageBox(). V tomto dialogovém okně se uživatele zároveň zeptáme, zda chce další podrobnosti, a pokud odpoví Ano, necháme vypsat zprávu uloženou v parametru handleru. K tomu použijeme metodu ShowException() třídy TApplication. (Zpráva doprovázející výjimku je ovšem anglicky.)

Metoda otevriClick() tedy bude mít tvar

```
void __fastcall Tokno::otevriClick(TObject *Sender)
{
  try
  {
              if(dialog_otevri -> Execute())
                  obrazek->Picture -> LoadFromFile(dialog_otevri -> FileName);
  }
  catch(Exception &E)
   {
     if(Application->MessageBox("Nepodařilo se otevřít grafický soubor. \n"
                "Chcete vědět více?", "Chyba", MB_YESNO) == IDYES)
       Application->ShowException(&E);
   }
}
```

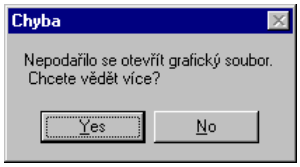
Čtení souboru s ošetřením možných výjimek

Poznámky

- ✧ Metoda TApplication::MessageBox() má tři parametry; první z nich je znakový řetězec, který chceme vypsat, druhý je řetězec určující titulek tohoto dialogového okna a třetí říká, jaká tlačítka a jaké ikony v tomto okně chceme. Příznak MB_YESNO např. předepisuje tlačítka Ano a Ne, příznak MB_OK samotné tlačítko OK, příznak MB_ICONQUESTION způsobí přidání ikony s otazníkem.
- ✧ Tato metoda vrací hodnotu odpovídající stisknutému tlačítku — např. IDYES odpovídá stisknutí tlačítka Yes.

[21] Pozor na možnost záměny s třídou exception deklarovanou v hlavičkovém souboru exception, která je společným předkem tříd pro práci s výjimkami ve standardní knihovně C++.

- ✧ Metoda MessageBox() je založena na stejnojmenné funkci z Windows API, takže podrobné informace o předdefinovaných hodnotách třetího parametru a o vracených hodnotách lze najít v nápovědě k této funkci.
- ✧ Prostředky pro toto dialogové okno jsou uloženy v operačním systému, takže pokud nepoužíváte českou verzi Windows, budou nápisy na tlačítkách anglicky — viz např. obr. 4.3.



Obr. 4.3 Dialogové okno s dotazem vytvořené pomocí funkce MessageBox()

Implicitní přípona

Jestliže v dialogovém okně pro výběr souboru neuvedeme příponu, bude vlastnost FileName obsahovat jméno souboru bez přípony a náš program ohlásí chybu (nepodařilo se otevřít soubor) i v případě, že soubor existuje. Protože jsme ale zvyklí, že mnohé programy si doplní příponu samy, upravíme program tak, aby nejprve zkontroloval, zda jméno souboru obsahuje příponu, a pokud ne, aby doplnil tu, která je nastavena jako implicitní.

Nejprve deklarujeme ve třídě Tokno datovou složku jmeno typu AnsiString a do ní uložíme jméno souboru přečtené z vlastnosti FileName dialogového okna. (Toto jméno budeme později potřebovat při ukládání obrázku, proto nám nestačí lokální proměnná.)

Pak zjistíme, zda jméno souboru obsahuje příponu. K tomu můžeme použít metodu LastDelimiter() třídy AnsiString, která vrátí index posledního výskytu zadaného znaku:

```
static AnsiString pripona[] = {"", ".bmp", ".ico", ".emf", ".wmf"};

if(!jmeno.LastDelimiter(".") && dialog_otevri->FilterIndex > 1)
  jmeno += pripona[dialog_otevri->FilterIndex-1];
```

Stejného výsledku ale dosáhneme pomocí funkce ExtractFileExt(), která prohledá zadané jméno souboru a vyjme z něj příponu.

```
if(ExtractFileExt(jmeno)==AnsiString(""))
  jmeno += pripona[dialog_otevri->FilterIndex-1];
```

Poznámky

- ✧ Připomeňme si, že FilterIndex je vlastnost dialogového okna TOpenPictureDialog, která určuje zvolený typ souboru. Na rozdíl od indexů polí jde od 1, nikoli od 0 — proto od něj musíme ve výrazu pripona[dialog_otevri->FilterIndex-1] odečíst 1.
- ✧ Řetězec vrácený funkcí ExtractFileExt() obsahuje příponu i s tečkou, tedy např. ".bmp".
- ✧ Vedle ExtractFileExt() najdeme v C++ Builderu další funkce umožňující rozklad úplného jména souboru na jednotlivé složky: ExtractFileName() vyjme z řetězce samotné jméno souboru, ExtractFilePath() vyjme část obsahující diskovou jednotku a cestu, ExtractFileDir() vyjme adresář a ExtractFileDrive() vyjme samotné označení diskové jednotky.
- ✧ Zkuste si sami upravit vytvořený program tak, aby v titulkové liště okna ukazoval jméno zobrazeného souboru bez cesty a diskové jednotky, ale s příponou.
- ✧ Pro další úpravy programu bude výhodné uložit si vedle jména souboru do zvláštní datové složky také příponu a filtr.
- ✧ Připojení implicitní přípony ke jménu souboru lze elegantněji řešit pomocí vlastnosti DefaultExt; vrátíme se k ní později v této kapitole.

Posuvníky

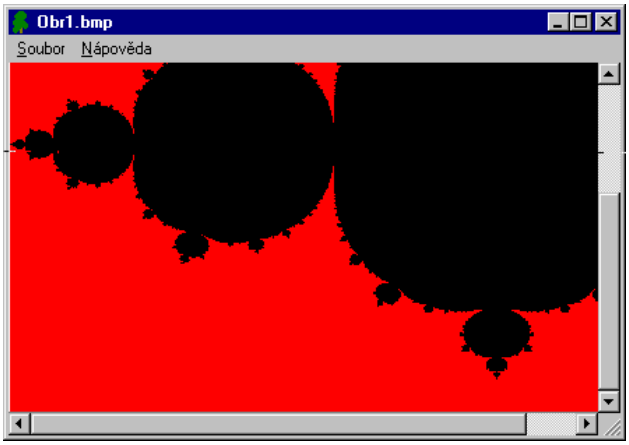
Abychom si mohli prohlédnout celý obrázek ve skutečné velikosti, připojíme k oknu posuvníky (v české literatuře najdete i termín „rolovací lišty“). Je to velice jednoduché, neboť formuláře mají tuto vlastnost již vestavěnu.

Nejprve nastavíme vlastnost AutoScroll formuláře okno na true (to je ve skutečnosti implicitní hodnota). Posuvníky jsou určeny vlastnostmi HorzScrollBar (vodorovný) a VertScrollBar (svislý). Každý z nich má řadu vlastností, z nichž asi nejdůležitější je Range (rozsah). Jediné, co musíme udělat, aby se u obrázku zobrazily posuvníky, je nastavit rozsah posuvníků. Rozsah vodorovného posuvníku musí obsahovat šířku obrázku, rozsah svislého výšku (v pixelech).

Rozměry obrázku jsou uloženy ve vlastnostech Width a Height vlastnosti Picture. Připojíme tedy do handleru Tokno::otevriClick() za čtení příkazy

```
okno->HorzScrollBar->Range = obrazek->Picture->Width;
okno->VertScrollBar->Range = obrazek->Picture->Height;
```

Jestliže nyní program přeložíme a spustíme, dostaneme již okno s fungujícími posuvníky (obr. 4.4).



Obr. 4.4 Okno s posuvníky

Abychom neztratili přehled, uvedeme výpis třídy Tokno a handleru Tokno::otevriClick():

```
class Tokno : public TForm
{
__published:  // IDE-managed Components
  TPanel *panel;
  TMainMenu *menu;
  TMenuItem *soubor;
  TMenuItem *napoveda;
  TMenuItem *otevri;
  TMenuItem *ulozjako;
  TMenuItem *zavri;
  TMenuItem *N1;
  TMenuItem *konec;
  TMenuItem *oprogramu;
  TImage *obrazek;
  TOpenPictureDialog *dialog_otevri;
  void __fastcall otevriClick(TObject *Sender);
private:          // User declarations
  AnsiString jmeno;   // Zde uložíme jméno souboru,
  AnsiString ext;     // jeho příponu
  AnsiString filtr;   // a filtr odpovídající typu souboru
public:           // User declarations
  __fastcall Tokno(TComponent* Owner);
};
```

Deklarace třídy Tokno

```
void __fastcall Tokno::otevriClick(TObject *Sender)
{
  static AnsiString pripona[5] = {"", ".bmp", ".ico", ".emf", ".wmf"};
  static AnsiString text[5] = {"Všechny soubory|", "Bitové mapy|", "Ikony|",
            "Vylepšené metasoubory|", "Metasoubory|",};
  if(dialog_otevri -> Execute()) // Pokud uživatel stiskl "Otevři"
  {
    try {
                    // Zjisti jméno zvoleného souboru, index filtru a příponu
          jmeno = dialog_otevri -> FileName;
          int index = dialog_otevri->FilterIndex-1;
          ext = pripona[index];
                    // Pokud jméno neobsahuje příponu, doplň ji
      if(ExtractFileExt(jmeno)==AnsiString(""))
```

```
                jmeno += ext;
                // Připrav filtr pro ukládání souboru
            filtr = text[index]+"*"+ext;

                // P e ti soubor s obr zkem
            obrazek->Picture -> LoadFromFile(jmeno); // Přečti soubor

                // Nastav rozsah posuvn k  a titulek okna
            okno->HorzScrollBar->Range = obrazek -> Picture -> Width;
            okno->VertScrollBar->Range = obrazek -> Picture -> Height;
            Caption = ExtractFileName(jmeno);
    }
    catch(Exception &E)
    {               // Pokud se čtení nepovede...
      if(Application->MessageBox("Nepodařilo se otevřít grafický soubor. \n"
                "Chcete vědět více?", "Chyba", MB_YESNO)== IDYES)
        Application->ShowException(&E);
    }
  }
}
```

Čtení obrázku

Uložení obrázku

Příkaz Soubor | Ulož jako... má za úkol uložit obrázek do souboru pod jiným jménem. Obrázek uložíme do souboru pomocí metody TPicture::SaveToFile(), které zadáme jako parametr jméno cílového souboru. Tato metoda ovšem neumí konvertovat různé typy grafických souborů, takže bitovou mapu musíme uložit jako bitovou mapu.metasoubor jako metasoubor atd.

Pro určení jména souboru a jeho adresáře použijeme komponentu SavePictureDialog, která obsahuje standardní dialogové okno Uložit jako.... Má podobné vlastnosti jako komponenta SavePictureDialog, takže náš výklad bude stručnější.

Nejprve tedy vložíme do vizuálního návrhu formuláře komponentu SavePictureDialog, pojmenujeme ji dialog_ulozjako a její vlastnosti Title dáme hodnotu Ulož jako. Její další vlastnosti ponecháme zatím beze změny, budeme je měnit až za běhu programu.

Pak ve vizuálním návrhu formuláře vybereme položku nabídky Soubor | Ulož jako.... Tím způsobíme, že prostředí vytvoří kostru handleru Tokno::ulozjakoClick() a přejde do něj.

V tomto handleru nejprve uložíme do vlastnosti FileName dialogového okna dialog_ulozjako aktuální jméno souboru, které máme uschované ve složce jmeno. Tím způsobíme, že nám dialogové okno Uložit jako... nabídne implicitně původní jméno souboru.

Pak uložíme do vlastnosti Filter předem připravený filtr (máme ho ve složce filtr). Tento filtr obsahuje pouze specifikaci odpovídající typu přečteného souboru — připomeňme si, že metoda SaveToFile() neumožňuje měnit typ obrázku.

Do vlastnosti DefaultExt uložíme příponu souboru, kterou máme ve složce ext. To způsobí, že pokud u jména souboru neuvedeme příponu nebo pokud uvedeme jinou příponu, připojí se automaticky tato.

Pak zavoláme metodu Execute(), a pokud vrátí true, tj. pokud uživatel stiskne tlačítko Uložit, soubor zapíšeme. Předtím ale zkontrolujeme, zda náhodou soubor zadaného jména neexistuje; K tomu použijeme funkci FileExists(). Pokud soubor existuje, zeptáme se uživatele, zda jej chce přepsat. [22]

Zdrojový kód metody Tokno::ulozjakoClick() může vypadat takto:

```
void __fastcall Tokno::ulozjakoClick(TObject *Sender)
{
            // Nejprve nastavíme vlastnosti dialogového okna
  dialog_ulozjako -> FileName = jmeno;
  dialog_ulozjako -> Filter = filtr;
  dialog_ulozjako -> DefaultExt = ext;

            // Pak zjist me jméno souboru
  if(dialog_ulozjako -> Execute())
  {
    jmeno = dialog_ulozjako -> FileName;

            // Pokud soubor existuje a u ivatel ho nechce p epsat, skon i
    if(FileExists(jmeno) &&
            !Application->MessageBox(
            ("Soubor "+jmeno+" existuje\nMám ho přepsat?").c_str(),
```

[22] Ve skutečnosti existuje jednodušší řešení: Stačí nastavit na true vlastnost ofOverwritePrompt, která je součástí vlastnosti Options komponenty TSavePictureDialog.

```
                "Upozornění", MB_YESNO | MB_ICONQUESTION))
                    return;
                // Jinak soubor ulož
    obrazek->Picture -> SaveToFile(jmeno);
  }
}
```

Uložení obrázku pod jiným jménem

Poznámky

- ✧ Použití implicitní přípony (vlastnosti DefaultExt) je poněkud zrádné. Jestliže např. nastavíme implicitní příponu *.bmp a zadáme jméno obr, uloží se obrázek do souboru obr.bmp. Jestliže ale zadáme jméno obr.wmf, uloží se do souboru jménem obr.wmf.bmp.
- ✧ Podívejme se na výraz

  ("Soubor "+jmeno+" existuje\nMám ho přepsat?").c_str(),

  který jsme použili jako první parametr metody TApplication::MessageBox(). Na první pohled vypadá možná nesrozumitelně, ale ve skutečnosti nejde o nic složitého. Součet

  "Soubor "+jmeno+" existuje\nMám ho přepsat?"

  vytvoří řetězec, který chceme zobrazit. Přitom využije přetížený operátor +, který umožňuje mj. spojovat instance třídy AnsiString a klasické céčkovské řetězce. Výsledkem je instance třídy AnsiString. Ovšem metoda MessageBox() má první parametr typu char*, tj. klasický céčkovský řetězec. Na vytvořený řetězec jsme tedy použili metodu AnsiStr::c_str(), která umožňuje přetypování typu AnsiString na char*. (Získaný ukazatel ale nelze použít ke změně obsahu instance typu AnsiString.)

Smíme skončit?

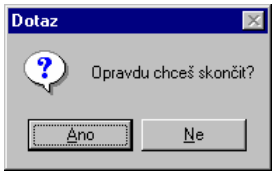
Slušný program se před ukončením zeptá, zda opravdu chceme skončit. Není nic jednoduššího; stačí ošetřit událost OnCloseQuery, která nastává vždy před ukončeném programu[23]. Handler FormCloseQuery() má vedle tradičního parametru Sender ještě parametr CanClose, který je typu bool a předává se odkazem. Přiřadíme-li mu hodnotu true, program opravdu skončí, uložíme-li do něj hodnotu false, bude pokračovat.

Pro dotaz použijeme opět metodu MessageBox() objektu Application, která vytvoří standardní okno s dotazem. Handler tedy bude vypadat např. takto:

```
void __fastcall Tokno::FormCloseQuery(TObject *Sender, bool &CanClose)
{
  if(Application -> MessageBox("Opravdu chceš skončit?", "Dotaz",
                                                    MB_YESNO | MB_ICONQUESTION)== mrYes)
    CanClose = true;
  else
    CanClose = false;
}
```

Handler, který si před ukončením programu vyžádá souhlas

Jestliže program nyní přeložíme, spustíme a pokusíme se ho ukončit (příkazem z nabídky nebo pomocí systémového menu), objeví se okno, které vidíte na obr. 4.1.



Obr. 4.5 Smí program skončit?

Stiskneme-li tlačítko Ano (nebo v našem případě Yes), program opravdu skončí, v opačném případě bude pokračovat.

Program v této podobě najdete na doprovodném CD v adresáři KAP04\01.

[23] Jde vlastně o reakci na zprávu WM_QUERYENDSESSION, kterou pošle operační systém před ukončením aplikace.

Obrázek zarovnaný do okna

Už víme, že vedle použití posuvníků máme ještě jednu možnost — transformovat obrázek tak, aby přesně vyplnil klientskou oblast okna. Toho dosáhneme, jestliže přiřadíme vlastnosti Stretch komponenty Image hodnotu true (implicitní hodnota je false). Upravíme tedy náš program tak, abychom mohli příkazem z nabídky přepínat mezi těmito dvěma způsoby zobrazení.

Nejprve přidáme do třídy Tokno datovou složku posuvniky typu bool. V ní si budeme ukládat informaci o aktuálním nastavení, tedy zda náš program používá posuvníky nebo zda upravuje velikost obrázku podle okna. V konstruktoru třídy Tokno jí přidělíme např. hodnotu true, takže náš program bude implicitně používat posuvníky. Pak do nabídky našeho programu přidáme položku Zobrazení s podřízenými položkami Přesná velikost a Podle okna; odpovídající ukazatele typu TMenuItem* pojmenujeme presne a podleokna.

V konstruktoru třídy Tokno „zaškrtneme“ odpovídající položku nabídky (nastavíme její vlastnost Checked na true):

```
__fastcall Tokno::Tokno(TComponent* Owner)
  : TForm(Owner), posuvniky(false)
{
  if(posuvniky)     // Zaškrtneme odpovídající položku menu
    presne -> Checked = true;
  else
    podleokna -> Checked = true;
}
```

Konstruktor třídy Tokno

Dále napíšeme funkce nastavPosuvniky() a zrusPosuvniky(), které budou měnit způsob zobrazování. První z nich, nastavPosuvniky(), změní „zaškrtnutí“ položek nabídky, zakáže úpravu velikosti a poměru stran (do vlastnosti obrazek -> Stretch uloží false), povolí posuvníky (uloží do AutoScroll hodnotu true) a definuje jejich rozsah. Druhá, zrusPosuvniky(), opět změní zaškrtnutí položek menu, povolí úpravu velikosti, zakáže posuvníky a nastaví jejich rozsah na 0.

Handlery ošetřující vybrání odpovídajících položek menu uloží do proměnné posuvniky hodnotu true nebo false a zavolají jednu z těchto funkcí:

```
void Tokno::nastavPosuvniky()
{
  presne -> Checked = true;              // Zaškrtni položku "Přesná velikost"
  podleokna -> Checked = false;          // Zruš zaškrtnutí "Podle okna"
  obrazek -> Stretch = false;          // Zakaž úpravu velikosti obrázku
  AutoScroll = true;                         // Povol posuvníky a nastav jejich rozsah
  okno->HorzScrollBar->Range = obrazek -> Picture -> Width;
  okno->VertScrollBar->Range = obrazek -> Picture -> Height;
}
void __fastcall Tokno::presneClick(TObject *Sender)
{
  posuvniky = true;                          // Nastav příznak způsobu zobrazení
  nastavPosuvniky();                         // a zavolej pomocnou funkci
}
void Tokno::zrusPosuvniky()
{
  presne -> Checked = false;             // Zaškrtni položku "Podle okna"
  podleokna -> Checked = true;           // Zruš zaškrtnutí "Přesná velikost"
  obrazek -> Stretch = true;             // Povol úpravu velikosti obrázku
  AutoScroll = false;                        // Zakaž posuvníky a zruš jejich rozsah
  okno->HorzScrollBar->Range = 0;
  okno->VertScrollBar->Range = 0;
}
void __fastcall Tokno::podleoknaClick(TObject *Sender)
{
  posuvniky = false;                         // Nastav příznak způsobu zobrazení
  zrusPosuvniky();                           // a zavolej pomocnou funkci
}
```

Funkce zajišťující přepínání způsobu zobrazení

Poslední úprava se bude týkat metody, která se stará o čtení souboru. I tato metoda by se měla řídit nastaveným způsobem zobrazování, a proto ji musíme lehce upravit. Po přečtení souboru zavoláme jednu z funkcí nastavPosuvniky() nebo zrusPosuvniky() a tím určíme způsob zobrazení. Zde si ukážeme jen část zdrojového textu:

```
void __fastcall Tokno::otevriClick(TObject *Sender)
{
  // ...
  if(dialog_otevri -> Execute())
  {
    try {
                // ...
      obrazek->Picture -> LoadFromFile(jmeno);
      if(posuvniky)
        nastavPosuvniky();
      else
        zrusPosuvniky();
      Caption = ExtractFileName(jmeno);
    }
    catch(Exception &E)
    {
                // ,,,
    }
  }
}
```

Upravená metoda otevriClick()

Po přeložení a spuštění uvidíme např. něco podobného jako na obr. 4.6. Poznamenejme, že takto můžeme změnit způsob zobrazení, nikoli však skutečnou velikost obrázku. Uložíme-li deformovaný obrázek příkazem Subor | Ulož jako..., uloží se bitová mapa, ikona nebo metasoubor v původní velikosti.

Obr. 4.6 Když se obrázek přizpůsobí rozměrům okna

Okno O programu (opakované použití)

Na závěr vytvoříme dialogové okno O programu. To už umíme, takže můžeme postupovat velmi rychle. Toto okno si uložíme pro budoucí použití v zásobníku objektů (Object Repository.)

- ✧ Připojíme k projektu nový formulář, který pojmenujeme např. Okno2. Zdrojový soubor s tímto oknem pojmenujeme např. oprog.cpp. Nastavíme vhodnou velikost, vlastnosti BorderStyle přiřadíme hodnotu bsDialog a vlastnosti Caption řetězec "O programu".
- ✧ Do okna vložíme text popisující smysl programu, informace o verzi, autorských právech atd. K tomu použijeme komponenty Label.
- ✧ Přidáme komponentu Image, kterou umístíme do pravé části okna. Do ní vložíme ikonu programu, logo firmy nebo jiný vhodný či méně vhodný obrázek.[24] (Vybereme v inspektoru objektů vlastnost Picture, klikneme na tlačítko se třemi tečkami vpravo od její hodnoty, a v dialogovém okně, které se objeví, stiskneme tlačítko Load. Vyhledáme obrázek a stiskneme tlačítko OK.) Vlastnosti Stretch přiřadíme hodnotu true.

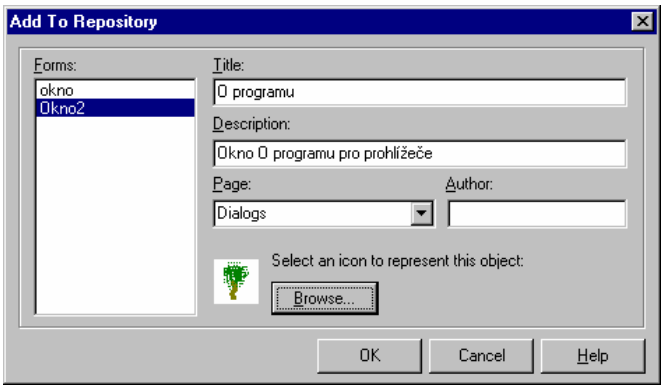
[24] Obrázek by měl již mít zhruba velikost, ve které ho budeme zobrazovat. Obrázek se ve spustitelném programu uloží celý a teprve při zobrazení okna se transformuje na zadanou velikost. Kdybychom připojili velký obrázek, zbytečně bychom zvětšili výsledný program.

Obr. 4.8 Dialogové okno O programu

✧ Do okna vložíme tlačítko — tj. komponentu Button — s nápisem OK. Vlastnostem Cancel a Default tohoto tlačítka přiřadíme hodnotu true (takže bude reagovat na klávesy ESC a ENTER) a vlastnosti ModalResult hodnotu mrOk, takže stisknutí tohoto tlačítka způsobí uzavření dialogového okna.

✧ Nakonec napíšeme handler, který toto okno zobrazí jako reakci na příkaz Nápověda | O programu.... Bude obsahovat jediný příkaz, Okno2 -> ShowModal();.

Výsledek vidíte na obr. 4.7. Nyní toto okno uložíme do zásobníku objektů. Klikneme pravým tlačítkem myši na jeho vizuální návrh a v příručním menu, které se objeví, zvolíme Add To Repository...; objeví se stejnojmenné dialogové okno (obr. 4.8).



Obr. 4.8 Ukládáme dialogové okno do zásobníku objektů

Zde v poli Forms vybereme okno, které chceme uložit. V poli Title zadáme název, pod kterým toto okno v zásobníku najdeme a v poli Page stránku zásobníku, na kterou ho chceme umístit; zvolíme stránku Dialogs. Nakonec pomocí tlačítka Browse konec zadáme ikonu, která bude toto dialogové okno reprezentovat, a stiskneme OK.

Uložené okno použijeme v následujícím oddílu.

4.2 Prohlížeč jako aplikace s rozhraním MDI

V tomto oddílu zkusíme další vylepšení: Rozšíříme prohlížeč tak, abychom si mohli najednou prohlížet několik obrázků. Vytvoříme tedy aplikaci s rozhraním MDI.

Co je to MDI

MDI je zkratka anglických slov Multiple Document Interface — rozhraní pro více dokumentů. Aplikace s tímto rozhraním umožňují pracovat s několika dokumenty najednou; typickým příkladem jsou předchozí verze textového procesoru MS Word.

Aplikace s rozhraním MDI má jedno hlavní okno (anglicky se nazývá frame window), které funguje jako kontejner, v němž jsou vnořena tzv. dětská okna (child windows). Tato dětská okna zobrazují dokumenty, se kterými pracujeme.

Dětská okna se chovají poněkud jinak než hlavní okno:

✧ Uzavření dětského okna nezpůsobí ukončení aplikace.

✧ Maximalizujeme-li dětské okno, vyplní celou klientskou oblast hlavního okna.

- ✧ Minimalizujeme-li dětské okno, zůstane jeho ikona (v novém uživatelském rozhraní část titulkové lišty) u spodního okraje hlavního okna.
- ✧ Dětská okna nemohou opustit klientskou oblast hlavního okna. Uvnitř hlavního okna je lze příkazem z nabídky uspořádat „schodovitě“ (do kaskády) nebo „jako dlaždice“.
- ✧ Dětská okna nemají vlastní nabídky.

Hlavní okno obsahuje základní nabídku. Obvykle v ní najdeme příkazy Soubor a Okno. Ostatní položky nabídky se zpravidla připojí až po otevření některého z dětských oken. Jedna z položek nabídky — zpravidla položka Okno — obsahuje seznam otevřených dětských oken a umožňuje přepínání mezi nimi.

Dětská okna mohou být všechna stejného typu nebo mohou být různých typů.

Poznámka

Ve skutečnosti je „okenní“ struktura aplikace s rozhraním MDI trochu složitější. Hlavní okno má jediné dětské okno, které se nazývá *MDI klient*. Toto okno sice nemá typické vlastnosti okna pod Windows (systémové menu, tlačítka v pravém horním rohu), ale je viditelné — pokrývá celou klientskou oblast hlavního okna. Dětská okna MDI jsou vnořena v klientovi MDI, nikoli v samotném hlavním okně.

Začínáme s MDI

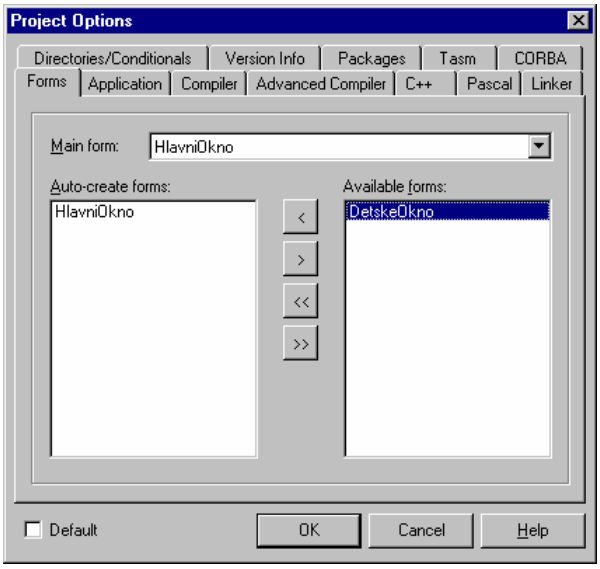
I když je vytváření aplikací s rozhraním MDI je v C++ Builderu jednoduché, začneme tím, že napíšeme jednoduchou aplikaci, která nebude dělat nic jiného než otevírat prázdná dětská okna. Teprve pak se vrátíme k našemu prohlížeči.

Vytvoříme nový projekt, který uložíme pod názvem MDI1. Modul s hlavním oknem aplikace uložíme pod názvem hlavni.cpp. Hlavní okno našeho programu pojmenujeme vtipně HlavniOkno a do jeho vlastnosti Caption vložíme řetězec "První pokus s MDI". Pak nastavíme vlastnost FormStyle na hodnotu fsMDIForm. Do definice třídy THlavniOkno v souboru hlavni.h přidáme soukromou složku kolik typu int. Tato proměnná bude sloužit jako počítadlo pro pojmenování dětských oken, a proto ji v konstruktoru třídy THlavniOkno inicializujeme hodnotou 0.

Dále vložíme do vizuálního návrhu hlavního okna komponentu MainMenu, kterou pojmenujeme HlavniMenu. Vytvoříme nabídku s položkami Soubor | Nový... , Soubor | Konec, Okno | Dlaždice a Okno | Kaskáda. Poslední dva příkazy budou mít za úkol uspořádat dětská okna. Odpovídající komponenty typu TMneuItem pojmenujeme soubor, novy, konec, okno, dlazdice a kaskada. V inspektoru objektů vyhledáme vlastnost WindowMenu hlavního okna a přidělíme jí hodnotu okno. Tím zajistíme, že do této nabídky připojí Windows seznam otevřených dětských oken.

Nyní do projektu přidáme nové okno (např. stisknutím tlačítka New Form na panelu nástrojů). Přidělíme mu jméno DetskeOkno, jeho vlastnost FormStyle nastavíme na fsMDIChild a vlastnost Caption vymažeme (uložíme do ní prázdný řetězec).

Dětských oken může být v běžícím programu více, nemusí v něm ale být žádné. Není tedy vhodné, aby se dětské okno na počátku vytvořilo automaticky. Vyvoláme proto dialogové okno Project | Options..., přejdeme na kartu Forms, v levém sloupci nadepsaném Auto-create forms (automaticky vytvářená okna) vybereme DetskeOkno a stisknutím tlačítka se znakem > ho přesuneme do sloupce Available forms (obr. 4.9).



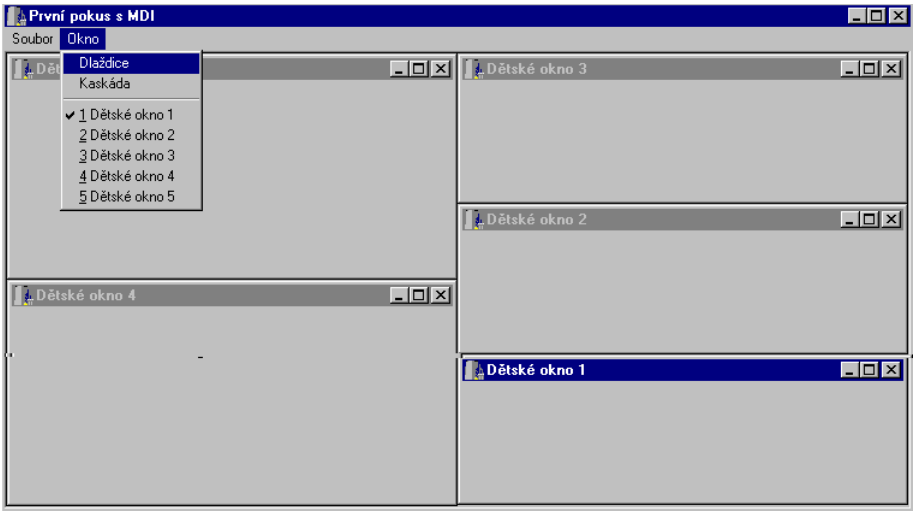
obr. 4.9 Určíme, které okno se má vytvořit automaticky a které si vytvoříme sami

Nyní naučíme program vytvořit dětské okno jako reakci na příkaz Soubor | Nový.... V handleru prostě vytvoříme novou instanci třídy TDetskeOkno, zobrazíme ji a do jejího titulku vložíme nápis "Dětské okno n" — viz následující výpis. Tím jsme téměř hotovi.

```
#include "hlavni.h"
#include "detske.h"

#pragma package(smart_init)
#pragma resource "*.dfm"
THlavniOkno *HlavniOkno;

__fastcall THlavniOkno::THlavniOkno(TComponent* Owner)
  : TForm(Owner), kolik(0)
{
}

void __fastcall THlavniOkno::novyClick(TObject *Sender)
{
  TDetskeOkno * okno = new TDetskeOkno(this);  // Vytvoříme okno
  okno ->Show();                                // a zobrazíme ho
  okno ->Caption = "Dětské okno " + AnsiString(++kolik);
}
```

Výpis souboru hlavni.cpp

Zbývá napsat handlery pro zbývající položky nabídky. Ukončit program již umíme, takže se podíváme na položky nabídky Okno.



Obr. 4.10 Náš první pokus o aplikaci s rozhraním MDI

Příkaz Dlaždice by měl uspořádat dětská okna v hlavní okně „jako dlaždice“, tedy stejně velká a jedno vedle druhého; příkaz Kaskáda by je měl dát všem oknům stejnou velikost a uspořádat je mírně posunutá jedno přes druhé. Nic z toho nemusíme na štěstí programovat, poslouží nám k tomu metody Tile() a Cascade() hlavního okna:

```
void __fastcall THlavniOkno::dlazdiceClick(TObject *Sender)
{
  Tile();
}
void __fastcall THlavniOkno::kaskadaClick(TObject *Sender)
{
  Cascade();
}
```

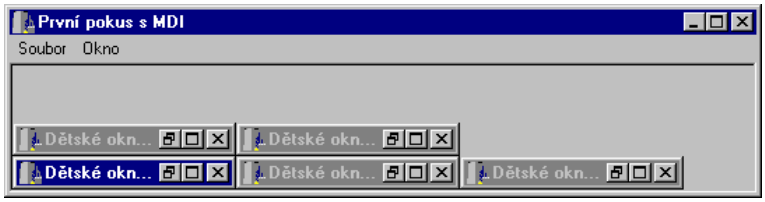
Metody pro uspořádání dětských oken

Nyní můžeme program přeložit a spustit. Výsledek ukazuje obr. 4.10.

Poznámka

Vedle metod Cascase() a Tile() máme k disposici ještě metodu ArrangeIcons(), která v hlavním okně uspořádá ikony minimalizovaných dětských oken.

Pokusy s tímto programem ale brzy ukáží jeden podivný nedostatek; pokusíme-li se zavřít dětské okno (např. dvojklikem na jeho systémové menu), nezmizí, ale minimalizuje se (obr. 4.11).



Obr. 4.11 Podivné chování dětských oken při zavírání

To je ovšem poněkud neobvyklé, a proto napíšeme handler, který ošetří událost OnClose dětského okna. Tento handler má dva parametry — Sender a Action. Druhý parametr se předává odkazem a určuje, co se má při uzavření dětského okna stát. Pro dětská okna má implicitní hodnotu caMinimize, která předepisuje, že se má dané okno minimalizovat. Další možné hodnoty jsou např. caHide, která způsobí, že se okno skryje, a caFree, která způsobí, že se okno uzavře a uvolní se alokovaná paměť. My použijeme poslední možnost.

```
void __fastcall TDetskeOkno::FormClose(TObject *Sender,
    TCloseAction &Action)
{
  Action = caFree;
}
```

Handler, který zajistí zrušení dětského okna při uzavření (soubor detske.cpp)

Potom se již budou dětská okna chovat podle očekávání.

Menu

Dětská okna v aplikacích MDI nemají nabídky (menu). To neznamená, že pro dětské okno nemůžeme nabídku definovat, ale že se tato nabídka na dětském okně nezobrazí; místo toho se sloučí s nabídkou hlavního okna.

Způsob sloučení závisí na vlastnosti GroupIndex (skupinový index) položek.

- ✧ Položka z nabídky dětského okna nahradí položku z nabídky hlavního okna se stejným skupinovým indexem.
- ✧ Pokožky s různým skupinovým indexem se zařadí za sebe podle hodnoty tohoto indexu.
- ✧ Nabídka, do které se vkládají jména otevřených dětských oken, by měla být nabídkou hlavního okna.

MDI – shrnutí

Shrňme kroky nezbytné k vytvoření aplikace s rozhraním MDI pomocí C++ Builderu:

1. Vytvoříme hlavní okno aplikace.
2. Pomocí komponenty MainMenu vytvoříme nabídku, která bude mj. obsahovat položku Okno (může se jmenovat i jinak, ale tento název je obvyklý).
3. Vlastnosti FormStyle hlavního okna přiřadíme hodnotu fsMDIForm, vlastnosti WindowMenu jméno nabídky Okno (té nabídky, ke které má operační systém připojovat seznam otevřených dětských oken).
4. Vytvoříme formulář pro dětské okno a v dialogovém okně Project | Options FormStyle hlavního okna přiřadíme hodnotu fsMDIForm na kartě Forms nastavíme, že se toto okno nemá vytvářet automaticky. Jeho vlastnosti FormStyle hlavního okna přiřadíme hodnotu fsMDIChild.
5. V handleru události OnClose předepíšeme chování dětského okna při uzavření, a to tak, že přiřadíme vhodnou hodnotu parametru Action. Zpravidla použijeme hodnotu caFree, která předepisuje uzavření okna a uvolnění paměti.
6. Napíšeme úsek programu, který se stará o vytváření dětských oken. Zpravidla jde o reakci na vybrání některé z položek menu. V něm pomocí operátoru new vytvoříme nové okno, zobrazíme je a naplníme daty.

Poznámky

- ✧ Při vytváření dětského okna jsme konstruktoru předali jako parametr this, tedy ukazatel na instanci hlavního okna. Tento parametr určuje vlastníka nově vytvořeného okna. Můžeme také použít ukazatel na instanci aplikace:

  ```
  TDetskeOkno * okno = new TDetskeOkno(Application);
  ```

  Ve většině případů to chování programu nijak neovlivní.
- ✧ Všimněte si, že jsme se nestarali o uvolnění dětského okna — nikde jsme nevolali operátor delete. O jeho uvolnění se program při uzavření tohoto okna postará automaticky.
- ✧ Tento program najdete na doprovodném CD v adresáři KAP04\03.

Vícedokumentový prohlížeč

Nyní využijeme, co jsme se v předchozím oddílu naučili, a konečně se pustíme do prohlížeče obrázků s rozhraním MDI. Položíme si tyto požadavky:

- ✧ Program po spuštění bude obsahovat jen prázdné hlavní okno a nezbytné nabídky. Další nabídky se doplní až po otevření prvního okna s obrázkem.
- ✧ Každý obrázek se zobrazí ve vlastním okně. Pro každý obrázek si můžeme předepsat, zda si jej budeme prohlížet pomocí posuvníků nebo zda má vyplnit celou klientskou oblast okna.

- ✧ Pro usnadnění práce poskytneme uživateli nástrojový panel s tlačítky, které budou představovat zkratky pro povely z nabídky. Pokud bude některý příkaz z nabídky nedostupný, musí být i odpovídající tlačítko „zašeděno“.
- ✧ Tlačítka budou nabízet bublinovou nápovědu. To znamená, že ponecháme-li na tlačítko po nějakou dobu kurzor myši, zobrazí se v „bublině“ u myši stručná informace o jeho významu.
- ✧ Prohlížeč bude zobrazovat pouze bitové mapy.

Hlavní okno

Nejprve vytvoříme potřebná okna. Otevřeme nový projekt, který nazveme prohl4. Soubor s hlavním oknem uložíme pod tradičním názvem hlavni.cpp.

Pak definujeme vlastnosti našeho hlavního okna: Ukazatel na něj pojmenujeme HlavniOkno (vlastnost Name), do titulkové lišty napíšeme opět Prohlížeč obrázků (vlastnost Caption) a stanovíme, že jde o hlavní okno aplikace MDI (FormStyle =).

Nabídka

Do vizuálního návrhu okna vložíme komponentu MainMenu, kterou pojmenujeme HlavniMenu. Dvojklikem na ni vyvoláme editor nabídek a vytvoříme položky Soubor | Otevři..., Soubor | Konec, Okno | Dlaždice, Okno | Kaskáda, Okno | Uspořádej ikony a Nápověda | O programu. Odpovídající položky pojmenujeme mn_soubor, mn_otevri, mn_konec, mn_okno, mn_dlazdice atd.

U položky Soubor (tj. u komponenty mn_soubor) ponecháme vlastnosti GroupIndex hodnotu 0, u položek okno a Nápověda (tj u komponent mn_okno a mn_napoveda) přiřadíme této vlastnosti hodnotu 5. To využijeme při slučování s nabídkou z dětského okna.

Protože při spuštění programu nebude otevřen žádný obrázek, nemá nabídka Okno smysl a mohlo by se zdát logické ponechat ji do dětského okna. Nicméně pro správnou funkci této nabídky je nezbytné, aby byla součástí hlavního okna, aby se při otevírání dětských oken neměnila. Proto jsme ji definovali zde. Abychom však vyhověli požadavkům, které jsme si položili v úvodu, přiřadíme na počátku její vlastnosti Visible hodnotu false. Všimněte si, že tato položka Okno nebude viditelná ani v době návrhu.

V inspektoru objektů přiřadíme vlastnosti WindowMenu hlavního okna hodnotu mn_okno.

Panel nástrojů

Pak do tohoto okna vložíme komponentu Panel, kterou pojmenujeme panel. Její vlastnosti Align přiřadíme hodnotu alClient, což způsobí zarovnání panelu k hornímu okraji okna, pod nabídku, a k oběma svislým okrajům. Jediné, co můžeme měnit, je výška panelu.

Dále vymažeme hodnotu vlastnosti Caption, takže panel neponese žádný nápis. Do tohoto panelu nyní vložíme 6 instancí komponenty SpeedButton, kterou najdeme na paletě Additional. To budou tlačítka pro otevření, uložení a uzavření obrázku, pro uzavření všech obrázků najednou a pro volbu, zda se má aktivní obrázek přizpůsobit velikosti okna nebo zda se mají použít posuvníky. Tyto komponenty pojmenujeme tl_otevri, tl_ulozjako, tl_zavri, tl_zavrivse, tl_scroll a tl_stretch.[25]

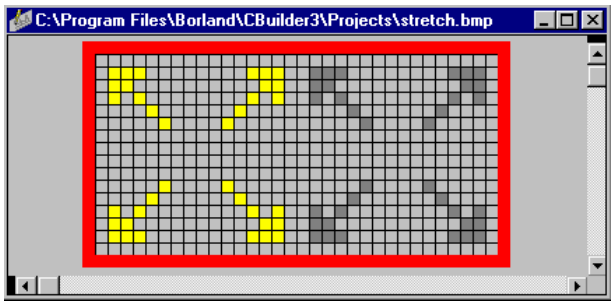
Obrázky pro jednotlivá tlačítka jsou určeny vlastností Glyph. Po vybrání této vlastnosti v inspektoru objektů a stisknutí tlačítka se třemi tečkami v poli s hodnotou se objeví dialogové okno pro Picture Editor pro výběr bitové mapy. V něm stiskneme tlačítko Load, vybereme vhodný obrázek a stiskneme OK.

Pro první čtyři obrázky můžeme využít předdefinovaných obrázků fileopen.bmp, fileclos.bmp, filasave.bmp a bookclose.bmp. Najdeme je spolu s řadou dalších v adresáři Program files\Common Files\Borland Shared\Images\Buttons a jsou volitelnou součástí instalace C++ Builderu.

Pro zbývající dvě tlačítka si obrázky vyrobíme sami. Použijeme k tomu editor obrázků, Image Editor, který známe už ze 2. kapitoly. Obrázek na tlačítko by měla být bitová mapa o rozměrech 32 × 16 bodů. V levé polovině tohoto obrázku bude ikona, která se na tlačítku zobrazí, pokud je aktivní (Enabled = true), v pravé polovině bude ikona, která se zobrazí, pokud je tlačítko neaktivní (Enabled = false). Obvykle se ikona pro aktivní a neaktivní tlačítko liší pouze barevným provedením — ikona pro

[25] Pokud se vám takovéto míchání češtiny a angličtiny v identifikátorech nezamlouvá, nic vám nebrání pojmenovat je jinak. Uvedené názvy ale najdete v programu na doprovodném CD.

aktivní tlačítko je barevná, ikona pro neaktivní tlačítko obsahuje týž obrázek v barvě tmavě šedé. Návrh bitové mapy pro tlačítko, které způsobí, že se obrázek přizpůsobí velikosti okna, ukazuje obr. 4.12.[26]



Obr. 4.12 Bitová mapa na tlačítko se skládá ze dvou částí

U prvního tlačítka (tl_otevri) ponecháme vlastnosti Enabled hodnotu true, u dalších čtyř jí přiřadíme hodnotu false, neboť před otevřením prvního obrázku nebudou mít odpovídající příkazy smysl.

Výšku panelu (komponenty panel), který tato tlačítka nese, upravíme tak, aby se do něj tlačítka právě vešla.

Bublinová nápověda

Ponecháme-li kurzor myši nějakou dobu nad některým tlačítkem, měla by se zobrazit bublinová nápověda. Obsah této nápovědy je určen vlastností Hint. Pro jednotlivá tlačítka přiřadíme této vlastnosti hodnoty " , "Z , "Ulož soubor z aktivního okna jako", "Zavři vše", " a " ". To je vše.

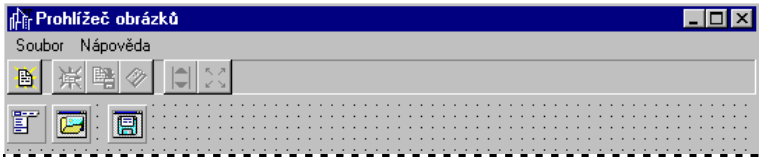
O tom, zda se bublinová nápověda pro dané tlačítko zobrazí, rozhoduje vlastnost ShowHint tlačítka (nebo obecně vizuální komponenty); nastavíme ji pro všechna tlačítka na true, takže se bude zobrazovat.

Poznamenejme, že zobrazování nebo nezobrazování bublinové nápovědy pro všechny vizuální komponenty lze řídit pomocí vlastnosti Application -> ShowHint; hodnota true, která znamená, že se bublinová nápověda bude zobrazovat, je opět implicitní.

Dialogová okna

Nakonec do vizuálního návrhu okna vložíme komponenty OpenPictureDialog a SavePictureDialog, které pojmenujeme dlg_otevri a dlg_ulozjako. Do vlastnosti Title první z nich uložíme řetězec "Otevři obrázek" a do vlastnosti Filtr uložíme filtr " , neboť nás zajímají pouze bitové mapy. Stejný filtr nastavíme i pro druhé dialogové okno. Vlastnosti DefaultExt obou oken přiřadíme hodnotu *.bmp. Ostatní vlastnosti budeme určovat za běhu programu.

Nyní by měl vizuální návrh hlavního okna vypadat podobně jako na obrázku 4.13.



Obr. 4.13 Vizuální návrh hlavního okna prohlížeče

Dětské okno

Naším dalším krokem bude návrh dětského okna. Stisknutím tlačítka New Form připojíme k projektu nový formulář, který v duchu nejlepších tradic této kapitoly pojmenujeme DetskeOkno. Jeho vlastnosti FormStyle přiřadíme hodnotu fsMDIChild (dětské okno MDI), hodnotu vlastnosti Caption vymažeme. (Do titulkové lišty vložíme za běhu programu jméno souboru s obrázkem.)

[26] Bitové mapy pro tato dvě tlačítka najdete na doprovodném CD v adresáři KAP04\04 pod názvy stretch.bmp a scroll.bmp.

V dialogovém okně Project | Options na kartě Forms určíme, že se tento formulář nemá vytvářet automaticky.

Do okna vložíme komponentu Panel, kterou opět pojmenujeme panel. Tento panel bude sloužit jako podklad pro obrázek. Jeho vlastnost Caption vymažeme, vlastnosti Align přiřadíme hodnotu alClient, takže vyplní celou klientskou oblast dětského okna.

Na panel položíme komponentu Image, kterou pojmenujeme obrazek. Její vlastnosti Align přiřadíme hodnotu alClient, takže vyplní celý panel.

Nyní bychom měli do tohoto okna ještě přidat nabídku (komponentu MainMenu), ale to odložíme na později — stejně jako vytvoření dialogového okna O programu.

Program

Nejprve si rozmyslíme, jaká data budeme za běhu programu potřebovat. Bude to počet existujících dětských oken a jméno souboru, ve kterém je uložen obrázek.

Počet aktivních dětských oken si uložíme[27] do datové složky kolik typu int ve třídě hlavního okna THlavniOkno; v konstruktoru ji inicializujeme hodnotou 0. Při vytvoření prvního dětského okna bude potřeba aktivovat tlačítka na panelu, po uzavření posledního je musíme deaktivovat; o to se postarají procedury PridejOkno() a ZrusOkno(). Vedle toho budeme potřebovat i procedury pro nastavení jednoho ze dvou možných způsobů zobrazení — s posuvníky nebo bez nich. Zkrácenou deklaraci třídy THlavniOkno ukazuje následující výpis.

```
class THlavniOkno : public TForm
{
__published
        // ... deklarace ukazatelů na komponenty a handlerů zde vynecháme
private:
  int kolik;                                    // Počítadlo dětských oken
public:
  __fastcall THlavniOkno(TComponent* Owner);
  void ZrusOkno();                              // Reakce na zrušení
  void PridejOkno();                            // a vytvoření dětského okna
  void NastavPosuvniky();                       // Zobrazení s posuvníku
  void NastavPrizpusobeni();                    // Zarovnání do okna
};
```

Zkrácený výpis třídy THlavniOkno

Jména otevřených souborů si budeme ukládat v dětských oknech, tj. v instancích třídy TDetskeOkno. V této třídě proto deklarujeme složku jmeno třídy AnsiString. Zároveň definujeme dvě přístupové funkce, nastavJmeno() a vratJmeno(). Zkrácenou deklaraci této třídy ukazuje následující výpis.

```
class TDetskeOkno : public TForm
{
__published:
  // ... deklarace ukazatelů na komponenty a handlerů zde vynecháme
private:
  AnsiString jmeno;                             // Jméno souboru s obrázkem
public:
  __fastcall TDetskeOkno(TComponent* Owner);
  void nastavJmeno(AnsiString a) {jmeno = a;};  // Přístupové funkce
  AnsiString vratJmeno() {return jmeno;}
};
```

Zkrácený výpis třídy TDetskeOkno

Otevření souboru

Dále se postaráme o reakci na příkaz z nabídky Soubor | Otevři. Zde nejprve pomocí dialogového okna dlg_otevri zjistíme jméno souboru a v případě potřeby k němu připojíme příponu .bmp. Pak vytvoříme dětské okno, zobrazíme ho a do jeho titulkové lišty vložíme jméno souboru (bez cesty a diskové jednotky). Jméno souboru uložíme do složky jmeno dětského okna. Pak se pokusíme obrázek přečíst, a pokud se to podaří, nastavíme režim zobrazení s posuvníky. Pokud se čtení nepodaří, vyvoláme dialogové okno s upozorněním a pak uzavřeme dětské okno.

```
void __fastcall THlavniOkno::mn_otevriClick(TObject *Sender)
```

[27] V tomto případě nemůžeme použít vlastnost MDIChildCount, neboť úpravy nabídky a tlačítek při zrušení posledního dětského okna budou probíhat v době, kdy toto okno bude ještě existovat — viz dále.

```
{
  AnsiString jmeno;                          // Pomocné proměnné
  TDetskeOkno *DO;
  if(dlg_otevri -> Execute())                // Spusť dialog pro vyhledání souboru
  {
    try                                                  // a pokud uživatel vybral soubor,
    {
                jmeno = dlg_otevri -> FileName;          // ulož si jeho jméno.
                DO = new TDetskeOkno(this);              // Vytvoř dětské okno
                DO -> Show();                                    // a zobraz ho.
                DO -> Caption = ExtractFileName(jmeno);
                DO -> nastavJmeno(jmeno);                // Nastav titulek okna
                DO -> obrazek -> Picture ->  LoadFromFile(jmeno); // Přečti obrázek
                NastavPosuvniky();                               // Nastav režim zobrazení
    }
    catch(Exception &e)              // Pokud se čtení nepodařilo,
    {                                                    // vypiš upozornění
     Application -> MessageBox (
                  ("Nepodařilo se otevřít obrázek " + jmeno).c_str(),
        "Chyba", MB_OK);
     DO -> Close();                  // a zavři dětské okno.
    }
  }
}
```

Výpis handleru, který reaguje na vybrání příkazu Soubor | Otevři…

Týž handler použijeme i v případě, že uživatel místo příkazu z nabídky stiskne tlačítko Otevři na nástrojovém panelu. (V inspektoru objektů si vybereme komponentu tl_otevri, přejdeme na kartu Events a v rozbalovacím seznamu u události OnClick zvolíme mn_otevriClick.)

Při otevření dětského okna je třeba zvýšit hodnotu proměnné kolik, a pokud jsme otevřeli první okno, také aktivovat tlačítka na panelu a zviditelnit položku Okno v hlavní nabídce. Podobně při uzavření dětského okna je potřeba snížit hodnotu proměnné kolik, a pokud jsme zavřeli poslední okno, také deaktivovat tlačítka na panelu a skrýt nabídku Okno. To budou mít na starosti funkce PridejOkno() a ZrusOkno().

```
void THlavniOkno::PridejOkno()
{
if(++kolik == 1)                           // Pokud jsme otevřeli první okno,
{
  mn_okno->Visible = true;           // zviditelni menu Okno
  tl_scroll -> Enabled = true;       // a aktivuj tlačítka
  tl_stretch -> Enabled = true;
  tl_zavri -> Enabled = true;
  tl_ulozjako -> Enabled = true;
  tl_zavrivse -> Enabled = true;
}
}

void THlavniOkno::ZrusOkno()
{
 if(!--kolik)                              // Pokud jsme zavřeli poslední okno,
{
  mn_okno->Visible = false;          // skryj menu Okno
  tl_scroll -> Enabled = false;      // a deaktivuj tlačítka
  tl_stretch -> Enabled = false;
  tl_zavri -> Enabled = false;
  tl_ulozjako -> Enabled = false;
  tl_zavrivse -> Enabled = false;
}
}
```

Funkce, které se starají o změny nabídky a aktivaci tlačítek

Nyní se rozhodneme, kdy a kde budeme tyto funkce volat. Mohlo by se zdát logické použít je v konstruktoru a v destruktoru, to ale vede za jistých okolností k problémům. (Nezapomínejme, že komponenty jsou napsány v Pascalu a proto se může lišit způsob volání konstruktorů předků; zkušenosti z C++ zde nemusí platit.) Proto funkci PridejOkno() zavoláme v handleru ošetřujícím událost OnCreate dětského okna a funkci ZrusOkno() v handleru[28] ošetřujícím událost OnClose(). Tyto handlery vidíte na následujícím výpisu.

[28] Událost OnClose nastane ještě v době, kdy rušené okno existuje. Proto používáme pomocnou proměnnou kolik a nikoli vlastnost MDIChildCount; viz též předchozí poznámku pod čarou.

```
void __fastcall TDetskeOkno::FormClose(TObject *Sender, TCloseAction &Action)
{
  Action = caFree;                    // Aby se dětské okno mohlo zavřít
  HlavniOkno -> ZrusOkno();    // Postarej se o tlačítka a menu
}
void __fastcall TDetskeOkno::FormCreate(TObject *Sender)
{
  HlavniOkno -> PridejOkno();       // Postarej se o tlačítka a menu
}
```

Ošetření událostí OnCreate a OnClose dětského okna

Dále musíme napsat funkce, které nastaví režim zobrazení — zda se použijí posuvníky nebo zda se rozměry obrázku přizpůsobí rozměrům klientské oblasti okna. To už umíme, takže rovnou uvedeme výpis odpovídajících funkcí.

```
void THlavniOkno::NastavPosuvniky()
{
  TDetskeOkno *DO = dynamic_cast<TDetskeOkno*>(ActiveMDIChild);
  if(DO)
  {
    DO -> obrazek -> Stretch = false;  // Zakaž přizpůsobení
    DO -> AutoScroll = true;                  // Povol posuvníku a nastav jejich rozsah
    DO -> HorzScrollBar -> Range = DO -> obrazek -> Picture -> Bitmap -> Width;
    DO -> VertScrollBar -> Range = DO -> obrazek -> Picture -> Bitmap -> Height;
  }
}
void THlavniOkno::NastavPrizpusobeni()
{
  TDetskeOkno *DO = dynamic_cast<TDetskeOkno*>(ActiveMDIChild);
  if(DO)
  {
    DO -> obrazek -> Stretch = true;       // Povol přizpůsobení
    DO -> AutoScroll = false;                 // Zakaž posuvníky
    DO -> HorzScrollBar -> Range = 0;    // a nastav jejich rozsah na 0
    DO -> VertScrollBar -> Range = 0;
  }
}
```

Funkce určující režim zobrazení

Protože se změna režimu má týkat aktivního dětského okna, zjistíme jeho adresu pomocí vlastnosti ActiveMDIChild. Ta ovšem obsahuje ukazatel typu TForm*, takže vrácenou hodnotu musíme nejprve přetypovat na ukazatel na TDetskeOkno. Pokud by se přetypování pomocí operátoru dynamic_cast nepodařilo, dostali bychom jako výsledek 0, a proto testujeme proměnnou DO v příkazu if.

Nyní již v našem programu funguje příkaz Soubor | Otevři a tlačítko Otevři. Po otevření prvního dětského okna se aktivují všechna tlačítka na nástrojovém panelu a přibude nabídka okno, do které systém připojuje seznam otevřených dětských oken.

Uzavření souboru

Handler, který se bude volat při stisknutí tlačítka Zavři, má za úkol zavřít aktivní dětské okno. Je velice jednoduchý:

```
void __fastcall THlavniOkno::tl_zavriClick(TObject *Sender)
{
  ActiveMDIChild -> Close();
}
```

Uzavření všech oken

I tato akce je velice jednoduchá. Ukazatele na dětská okna jsou uložena v poli MDIChildren a jejich počet je v MDIChildCount:

```
void __fastcall THlavniOkno::tl_zavrivseClick(TObject *Sender)
{
  for(int i = 0; i < MDIChildCount; i++)
    MDIChildren[i] -> Close();
}
```

Uložení souboru pod jiným jménem

Další tlačítko by mělo sloužit k uložení souboru pod jiným jménem. Nejprve si zjistíme ukazatel na aktivní dětské okno a původní jméno souboru, které nabídneme jako počáteční hodnotu v dialogovém okně dlg_ulozjako. V titulku tohoto dialogového okna uvedeme i jméno souboru, samozřejmě bez cesty.

Pokud při ukládání zjistíme, že soubor udaného jména existuje, zeptáme se, zda ho uživatel chce přepsat.

```
void __fastcall THlavniOkno::tl_ulozjakoClick(TObject *Sender)
{
  TDetskeOkno *DO = dynamic_cast<TDetskeOkno*>(ActiveMDIChild);
  if(DO)                              // Ukazatel na aktivní dětské okno
  {                                   // Zjistíme původní jméno souboru
    AnsiString jmeno = DO -> vratJmeno();          // a použijeme ho v titulku
    dlg_ulozjako -> Title = "Ulož " + ExtractFileName(jmeno) + " jako...";
    dlg_ulozjako -> FileName = jmeno;              // Nabídneme původní jméno
    if(dlg_ulozjako -> Execute())
    {                                                          // Pokud uživatel stiskl Uložit
      jmeno = dlg_ulozjako -> FileName;   // zjistíme nové jméno a pokud
                                                               // soubor existuje, zeptáme se,
                                                               // zda ho uživatel chce přepsat
      if(FileExists(jmeno) && Application -> MessageBox(
        (jmeno + "existuje.\nPřepsat?").c_str(),
        "Dotaz", MB_YESNO | MB_ICONQUESTION))
          return;
      DO->obrazek->Picture->SaveToFile(jmeno);// Ulož soubor
      DO->Caption = ExtractFileName(jmeno);
      DO->nastavJmeno(jmeno);                  // Ulož nové jméno do titulku okna
    }                                                          // a do složky jmeno v instanci
  }                                                                                    // třídy dětského okna
}
```

Výpis handleru, který se stará o uložení souboru pod jiným jménem

Nastavení režimu zobrazení

Poslední dvě tlačítka mají na starosti změnu režimu zobrazení v aktivním okně. Kód handlerů, které se postarají o událost OnClick, bude opět velmi jednoduchý — zavolá již hotové funkce NastavPosuvniky() a NastavPrizpusobeni().

```
void __fastcall THlavniOkno::tl_scrollClick(TObject *Sender)
{
  NastavPosuvniky();
}
void __fastcall THlavniOkno::tl_stretchClick(TObject *Sender)
{
  NastavPrizpusobeni();
}
```

Handlery, které se starají o změnu režimu zobrazení

Uspořádání dětských oken

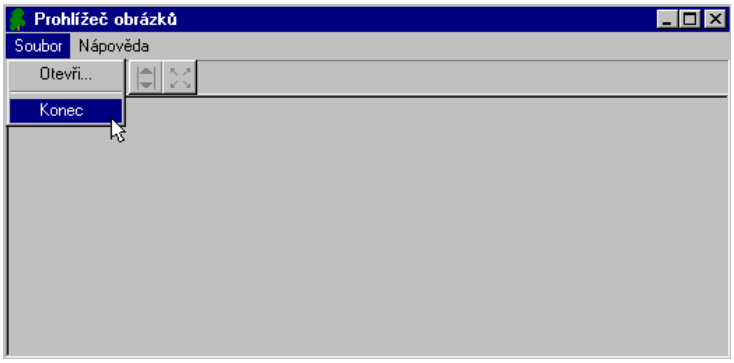
Zbývá ještě napsat handlery pro položky Kaskáda, Dlaždice a Uspořádej ikony v nabídce Okno. První z nich bude volat metodu Cascade(), druhý Tile() a třetí ArrangeIcons(). To už známe, a proto zde nebudeme uvádět výpisy.

Nabídka dětského okna

Nyní si ukážeme, jak lze využít nabídky definované v dětském okně. Do dětského okna vložíme komponentu MainMenu, kterou pojmenujeme DetskeMenu. Tato nabídka bude obsahovat položku Soubor s podřízenými položkami Otevři..., Ulož jako..., Zavři a Konec a položku Zobrazení s podřízenými položkami Přizpůsob se oknu a Použij posuvníky. (Jde vesměs o operace, které jsou k disposici prostřednictvím tlačítek na panelu.) Odpovídající komponenty pojmenujeme mn_soubor, mn_otevri atd.

Položce Soubor, tj. komponentě mn_soubor, ponecháme skupinový index 0 (připomeňme si, že jde o vlastnost GroupIndex). Protože stejný skupinový index má položka Soubor nabídky hlavního okna, znamená to, že po otevření prvního dětského okna nahradí tato položka položku Soubor z nabídky hlavního okna.

Položce Zobrazení, tj. komponentě mn_zobrazeni, přidělíme skupinový index 1. Protože takový index žádná položka nabídky hlavního okna nemá (u položek Okno a Nápověda jsme hodnotu skupinového indexu nastavili na 5), zařadí se položka Zobrazení mezi položky Soubor a Okno.

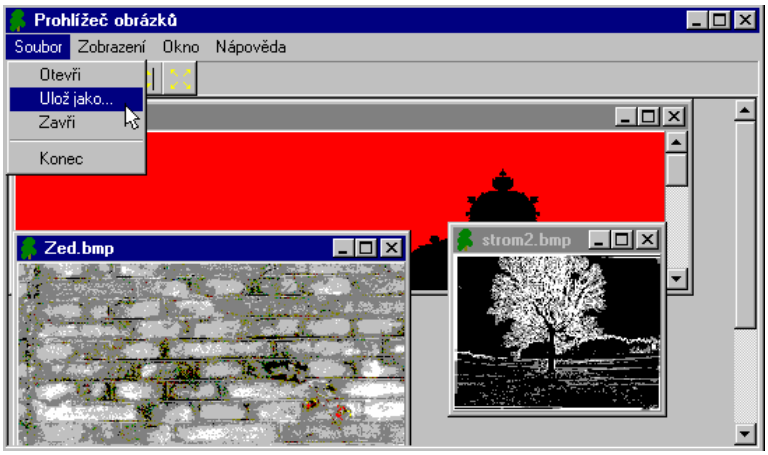


Obr. 4.14 Když není otevřeno žádné dětské okno

Handlery pro tyto položky nabídky prostě zavolají handlery pro odpovídající tlačítka, např.

```
void __fastcall TDetskeOkno::mn_prizpusobClick(TObject *Sender)
{
  HlavniOkno -> tl_stretchClick(Sender);
}
```

Vzhled programu v situaci, kdy není otevřeno žádné dětské okno, ukazuje obr. 4.14. Týž program po otevření několika dětských oken ukazuje obr. 4.15.



Obr. 4.15 Týž program, je-li několik oken otevřených

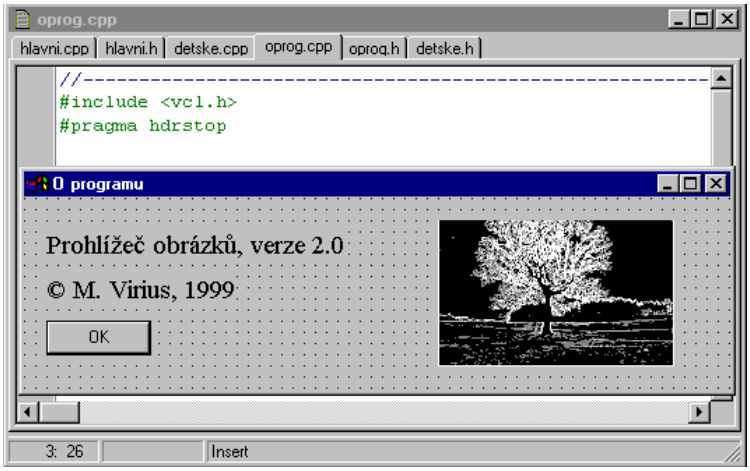
Dialogové okno O programu

Poslední, co nám zbývá, je vytvořit dialogové okno O programu, které se zobrazí po vybrání položky Nápověda | O programu z nabídky. Toto okno jsme si ale již vytvořili a uložili v zásobníku objektů, takže to půjde rychle.

V nabídce C++ Builderu použijeme příkaz File | New. Objeví se zásobník objektů, ve kterém přejdeme na kartu Dialogs (právě sem jsme si vytvořené okno uložili). Ve spodní části tohoto okna jsou tři přepínače, Copy, Inherit a Use. My zvolíme první z nich, neboť zbývající dvě znamenají, že případné změny v původním oknu by se promítly i do okna v našem projektu. Prostředí připojí k našemu programu zdrojový soubor a vizuální návrh tohoto okna (obr. 4.16). Ve vizuálním návrhu opravíme číslo verze, nic jiného není třeba, a uložíme soubor pod jménem oprog.cpp. Třída tohoto okna se jmenuje TOknoOProgramu, samotná instance OknoOProgramu. Jediné, co musíme udělat, je napsat handler, který toto okno zobrazí:

```
void __fastcall THlavniOkno::mn_oprogramuClick(TObject *Sender)
{
  OknoOProgramu -> ShowModal();
}
```

Tím jsme hotovi a můžeme prohlížeč používat.



Obr. 4.16 Dialogové okno O programu převzaté ze zásobníku objektů

5. Kreslítko

V této kapitole se pustíme do složitějšího projektu — vytvoříme jednoduchý grafický editor, kterému budeme říkat *kreslítko*. Obrázky v něm budeme kreslit myší: Na nástrojovém panelu zvolíme sílu a barvu čáry, umístíme kurzor myši do počátečního bodu, stiskneme levé tlačítko a kreslíme. Vytvořený obrázek půjde uložit ve formátu, který bude možno tímto programem znovu přečíst, nebo jako bitovou mapu.

5.1 Obrázek se skládá z čar

Nejprve si ujasníme, jak náš obrázek vznikne a jak ho budeme ukládat v paměti počítače a do souboru.

Při stisknutí tlačítka myši nastane v programu událost OnMouseDown. Z parametrů handleru se dozvíme mj. souřadnice bodu, ve kterém byl v okamžiku stisknutí tlačítka kurzor myši; to bude počáteční bod kreslené čáry.

Při tažení myši nastane v programu událost OnMouseMove. Pohybujeme-li myší déle, nastává tato událost v pravidelných intervalech a z parametrů hadleru se dozvídáme souřadnice bodů, ve kterých byl při nich kurzor myši. To budou další body kreslené čáry. (Ovšem pozor, nás bude zajímat pouze pohyb myši, při kterém je stisknuto pravé nebo levé tlačítko myši.)

Při uvolnění tlačítka myši nastane v programu událost OnMouseUp. Z parametrů handleru se opět dozvíme souřadnice bodu, ve kterém byl v okamžiku uvolnění tlačítka kurzor myši; to bude koncový bod kreslené čáry.

Celý obrázek se bude skládat z takovýchto lomených čar. (Ovšem intervaly, ve kterých nastává událost OnMouseMove, jsou tak krátké, že nebude problém nakreslit čáru, která bude vypadat hladká — alespoň pokud použijeme silnější pero.)

K uložení obrázku v paměti tedy potřebujeme nějaký kontejner — datovou strukturu, do které bychom mohli ukládat jednotlivé čáry. Vzhledem k tomu, že předem nevíme, kolik jich bude, nehodí se *pole*; použijeme tedy *spojový seznam*. Definujeme třídu Tobrazek, která bude obsahovat celkový počet čar v obrázku a jejich seznam.

Dále musíme navrhnout způsob uložení jednotlivých čar. Protože jde o lomenou čáru, která je určena svými vrcholy, stačí si uložit tyto vrcholy, tedy seznam bodů, sílu a barvu čáry. Pro uložení bodů použijeme opět spojový seznam.

Seznam naštěstí nemusíme programovat; ve standardní šablonové knihovně (STL) jazyka C++, která je součástí instalace C++ Builderu, je mj. šablona list<T>, podle které můžeme vytvořit obousměrný seznam s prvky typu T. Vedle toho tu najdeme šablony dalších obecně užitečných kontejnerů, jako jsou fronty, zásobníky, asociativní pole, množiny atd. Šablona seznamu je v hlavičkovém souboru list[29].

Bod, čára, obrázek

Navrhneme tedy datové typy, které použijeme k implementaci obrázku, a uložíme je v souboru cara.h; implementace jejich metod bude v souboru cara.cpp. Začneme u třídy reprezentující bod, tedy dvojici celých čísel. Použijeme konvencí C++ Builderu a nazveme ji Tbod:

```
struct Tbod
{
  int x, y;
  Tbod(int xx=0, int yy=0): x(xx), y(yy){}
};
```

Deklarace třídy Tbod (soubor cara.h)

Dále navrhneme třídu Tcara reprezentující čáru. Ta bude obsahovat seznam bodů, sílu a barvu čáry a počet bodů, které čáru tvoří. Tato třída bude potřebovat konstruktory, metodu pridej(), která k čáře připojí nový bod, metodu nakresli(), která tuto čáru nakreslí na obrazovku, a metodu uloz(), která zapíše čáru do daného souboru.

[29] Jako všechny standardní hlavičkové soubory jazyka C++ postrádá příponu .h.

```
#include <list>
#include <cstdio>
using namespace std;
class Tcara
{
   int pero;                          // Síla čáry
   TColor barva;                      // Barva čáry
   int delka;                         // Počet bodů v čáře
   list<Tbod> body;                   // Seznam bodů tvořících čáru
public:
   Tcara(TColor _barva, int _pero)
               : barva(_barva), pero(_pero), delka(0)
               {}
   Tcara(FILE* F);                    // Čtení čáry jako konstruktor
   void pridej(Tbod& b)               // Přidání nového bodu do čáry
               { body.push_back(b); delka++; }
   void uloz(FILE * F);               // Uložení čáry do souboru
   void nakresli(TCanvas *C);         // Nakreslení čáry na obrazovku
};
```

Deklarace třídy Tcara (soubor cara.h)

Poznámky

- ✧ Všimněte si, že místo „obyčejné" metody pro čtení čáry ze souboru jsme deklarovali konstruktor, který přečte uložená data a pak vytvoří odpovídající čáru.
- ✧ Metoda list<T>::push_back() připojí kopii parametru na konec seznamu.
- ✧ Destruktor seznamu při zániku zavolá destruktory uložených objektů.
- ✧ Šablona list<>, podobně jako ostatní jména ze standardní knihovny jazyka C++, leží v prostoru jmen std. Abychom tedy nemuseli psát std::list<> apod., použili jsme na počátku tohoto hlavičkového souboru direktivu using.
- ✧ Při ukládání do souboru používáme prostředky jazyka C, které jsou v hlavičkovém souboru stdio.h. V C++ je tento hlavičkový soubor k disposici pod jménem cstdio. (Všechny standardní hlavičkové soubory jazyka C jsou v C++ k disposici pod jmény, která vzniknou z původního označení vypuštěním přípony .h a připojením před jméno. Můžeme se na ně ale odvolávat i pod původním jménem.)

Nyní už můžeme navrhnout typ Tobraz, který bude reprezentovat celý obrázek. Tato třída bude kromě seznamu čar obsahovat také údaj o počtu čar v obrázku.

```
class Tobraz
{
   int kolik;                         // Počet čar v obrázku
   list<Tcara> cary;                  // Seznam čar tvořících obrázek
public:
   Tobraz(): kolik(0){};
   void nakresli(TCanvas *C); // Nakresli obrázek
   void pridej(Tcara* x)          // Přidej novou čáru do obrázku
               { cary.push_back(*x); kolik++; }
   void uloz(FILE*);                  // Ulož  obrázek do souboru
   void precti(FILE*);                // Přečti obrázek ze souboru
};
```

Deklarace třídy Tobraz (soubor cara.h)

Nyní se podíváme na implementaci jednotlivých metod v souboru cara.cpp. Začneme u třídy Tcara.

Metody třídy Tcara

Kreslení čáry

Metoda Tcara::nakresli() dostane jako parametr plátno (canvas), do kterého bude kreslit. Nastaví sílu a barvu pera a pak v cyklu projde body uložené v seznamu a spojí je čarou.

```
void Tcara::nakresli(TCanvas *C)
{
 C -> Pen -> Width = pero;                  // Tloušťka čáry
 C -> Pen -> Color = barva;                 // Barva čáry
```

```
 list<Tbod>::iterator i = body.begin();      // Počáteční bod
 C -> MoveTo((*i).x, (*i).y);                // Přejdi do počátečního bodu
 for(i = body.begin(); i != body.end(); i++)
   C -> LineTo((*i).x, (*i).y);              // Vykresli čáru
}
```

Kreslení čáry na základě uložených dat (soubor cara.cpp)

První dva příkazy nastavují vlastnosti pera, kterým se bude čára kreslit; to známe už ze druhé kapitoly.

Iterátor na seznamu

Při práci se seznamem používáme iterátory. Protože jde o pojem, který není všeobecně známý, zastavíme se u něj.

Na seznam se můžeme dívat jako na posloupnost prvků; v tom se podobá poli. Na rozdíl od pole ovšem nevíme, kolik těch prvků je.

Při práci s polem obvykle používáme ukazatele. Jestliže ukazatel `u` ukazuje na nějaký prvek pole, pak `u++` ukazuje na prvek, který za ním následuje, a `u--` ukazuje na prvek, který mu předchází. Pomocí operace `u++` tedy můžeme projít celé pole prvek po prvku.

Iterátor je jakési zobecnění ukazatele — je to datová struktura, která umožňuje pracovat s prvky seznamu. Jestliže iterátor `i` ukazuje na nějaký prvek seznamu, pak `*i` představuje data uložená v tomto prvku, `i++` bude ukazovat na následující prvek a `i--` na předcházející prvek.

Třída iterátoru je definována ve třídě seznamu, a proto musíme iterátor pro náš seznam deklarovat zápisem

```
list<Tbod>::iterator i = body.begin();
```

Zde jsme mu zároveň přidělili počáteční hodnotu — iterátor ukazující na první prvek seznamu `body`.

Protože pro tento iterátor není definován operátor `->`, museli jsme ke složkám bodů v seznamu `body` přistupovat zápisem `(*i).x` a `(*i).y`.

Iterátory na seznamu můžeme mezi sebou porovnávat pomocí přetížených operátorů `==` a `!=` Metoda `list<T>::begin()` vrací iterátor ukazující *na* první prvek našeho seznamu, metoda `list<T>::end()` vrací iterátor ukazující *za* poslední prvek seznamu. To nám umožňuje použít iterátor i jako parametr cyklu. Např. v příkazu

```
for(i++; i != body.end(); i++) C -> LineTo((*i).x, (*i).y);
```

nakreslíme postupně spojnici prvního bodu s druhým, druhého s třetím atd. Přitom předpokládáme, že před vstupem do tohoto cyklu ukazuje iterátor `i` na první prvek seznamu — proto jsme jako inicializaci použili `i++`.

Uložení čáry do souboru

Metoda `Tcara::uloz()` zapíše do souboru délku, sílu pera (šířku čáry), barvu a jednotlivé body. Protože nám jde o neformátovaný binární zápis, použijeme funkci `fwrite()`. Při ukládání seznamu použijeme opět iterátoru.

```
void Tcara::uloz(FILE * F)
{
 fwrite(&delka, sizeof(delka), 1, F);          // Zapiš počet bodů,
 fwrite(&pero, sizeof(pero), 1, F);            // šířku pera,
 fwrite(&barva, sizeof(barva), 1, F);          // barvu
 for(list<Tbod>::iterator i = body.begin(); i != body.end(); i++)
 {                                                                  // a jednotlivé body
   Tbod b = *i;                                                 // uložené v seznamu "body"
   fwrite(&b, sizeof(b), 1, F);
 }
}
```

Uložení čáry do souboru (soubor cara.cpp)

Čtení čáry ze souboru

Čtení je podobné jako ukládání, pouze použijeme funkci `fread()`. Protože jako první přečteme délku čáry, můžeme ke čtení jednotlivých bodů použít cyklus se známým počtem kroků. Vzhledem k tomu, že jsme čtení svěřili konstruktoru, bude výsledkem nová instance obsahující načtená data.

```
Tcara::Tcara(FILE* F)
{
 fread(&delka, sizeof(delka), 1, F);
 fread(&pero, sizeof(pero), 1, F);
 fread(&barva, sizeof(barva), 1, F);
 for(int i = 0; i < delka; i++)
 {
   Tbod b;
   fread(&b, sizeof(b), 1, F);
   body.push_back(b);
 }
}
```

Konstruktor, který přečte data ze souboru a vytvoří odpovídající instanci typu Tcara (soubor cara.cpp)

Metody třídy Tobraz

Nakreslení obrázku

Protože obrázek je seznam čar, stačí, když metoda Tobraz::nakresli() projde v cyklu celý seznam a každou čáru, která je v něm uložena, nakreslí. Přitom opět použijeme iterátor na seznamu. Parametrem této metody je ukazatel na plátno (canvas), do kterého chceme kreslit.

```
void Tobraz::nakresli(TCanvas *C)
{
  for(list<Tcara>::iterator i = cary.begin();
     i != cary.end(); i++)
       (*i).nakresli(C);
}
```

Kreslení celého obrázku (soubor cara.cpp)

Uložení celého obrázku do souboru

U souborů obsahujících binární data je poměrně obvyklé, že začínají „hlavičkou" — záznamem, který jednoznačně identifikuje typ souboru. My použijeme jako hlavičku znakový řetězec "Kreslítko 1.0". (Abychom si zbytečně nekomplikovali život, bude stejný pro všechny verze.)

```
static char* text = "Kreslitko 1.0";

void Tobraz::uloz(FILE* F)
{
 fwrite(text, AnsiString(text).Length(), 1, F);   // Zapiš hlavičku,
 fwrite(&kolik, sizeof(kolik), 1, F);              // počet čar
 for (list<Tcara>::iterator i = cary.begin();
     i != cary.end(); i++)
       (*i).uloz(F);                                    // a jednotlivé čáry
}
```

Uložení kresby do souboru (soubor cara.cpp)

Tato metoda nejprve zapíše hlavičku, pak počet čar v souboru a nakonec v cyklu jednotlivé čáry. Přitom využije metodu uloz() třídy Tcara.

Poznámka

Metoda AnsiString::Length() vrátí délku řetězce. Třída AnsiString obsahuje konstruktor s parametrem typu char*, takže zápis AnsiString(text) představuje vlastně přetypování céčkovského řetězce na AnsiString a zápis AnsiString(text).Length() pak zjistí délku (počet znaků) řetězce text. Stejně dobře jsme ke zjištění délky řetězce mohli použít funkci strlen() známou z jazyka C.

Čtení obrázku ze souboru

Čtení obrázku bude o něco složitější.

```
void Tobraz::precti(FILE* F)
{
 char xtext[20]={0,};                                    // Pomocné pole
 fread(xtext, AnsiString(text).Length(), 1, F);  // Přečteme hlavičku
 if(AnsiString(xtext) != AnsiString(text))               // a porovnáme ji
```

```
{                                                                    // se vzorem - a pokud nesouhlasí,
  Application -> MessageBox(                                  // vypíšeme zprávu a skončíme
                "Daný soubor není ve formátu Kreslítko 1.0" ,
                "Problém" , MB_OK);
  return;                                                            // se čtením
}
int k=0;                                                             // Pomocná proměnná
fread(&k, sizeof(k), 1, F);                                 // Přečteme počet čar
for (int i = 0;i < k; i++)                                  // Čteme jednotlivé čáry
     pridej(new Tcara(F));                                  // a připojujeme je k obrázku
}
```

Čtení obrázku ze souboru (soubor cara.cpp)

Poznámka k ladění

Všechny naše předchozí programy začínaly vytvořením okna aplikace, takže bylo od počátku jasné, jak takový program ladit. Tentokrát je ale situace jiná. Při ladění tříd Tobraz a Tcara potřebujeme pomocný program, na kterém bychom si mohli vyzkoušet jejich funkčnost.

Asi nejjednodušší je vytvořit program s prázdným oknem a testování našich tříd svěřit např. handleru, který bude ošetřovat událost OnClick. Můžeme si také vytvořit konzolovou aplikaci a v ní tyto třídy odladit.

5.2 Okno kreslítka

Nyní se konečně pustíme do našeho kreslítka. Vytvoříme nový projekt aplikace pro Windows a uložíme ho pod názvem kresl.cpp. Hlavní okno programu pojmenujeme tradičně HlavniOkno a do titulkové lišty (vlastnost Caption) napíšeme Kreslítko.

Do definice třídy THlavniOkno přidáme složku barva typu TColor, ve které si budeme uchovávat nastavenou barvu čáry, a složku sila typu unsigned, ve které si budeme ukládat aktuální tloušťku pera.

Pak do tohoto okna vložíme komponentu MainMenu, kterou pojmenujeme HlavniMenu. Tato nabídka bude mít položku Soubor s podřízenými položkami Nový, Otevři..., Ulož, Ulož jako a Konec. Kromě toho bude mít položku Nápověda s podřízenou položkou O programu.... Odpovídající komponenty pojmenujeme mn_soubor, mn_novy atd.

Dále vložíme do hlavního okna komponentu Panel, kterou uložíme pod jménem pa_paleta, neboť bude obsahovat nástroje pro volbu barvy. Hodnotu vlastnosti Caption vymažeme, takže panel neponese žádný nápis, a hodnotu vlastnosti Align nastavíme na alTop, takže se panel zarovná s horním okrajem okna a s bočními okraji. Myší ho roztáhneme tak, aby zabíral většinu plochy okna; to nám usnadní práci s komponentami, které na něj budeme vkládat.

Výběr barvy

Pro výběr nejběžnějších barev lze použít komponentu CColorGrid (barevná mřížka), kterou najdeme na paletě Samples. Pojmenujeme ji barvy.

Tato mřížka se skládá ze 16 barevných polí. Klikneme-li na některé z nich levým tlačítkem myši, vybereme barvy popředí, klikneme-li pravým tlačítkem myši, vybereme barvu pozadí. Vybrané barvy zjistíme jako hodnoty vlastností ForegroundIndex a BackgroundIndex; to jsou ale bohužel čísla v rozmezí 0 – 15, nikoli skutečné hodnoty barvy. Pomůžeme si tedy tím, že vytvoříme pole barevných hodnot odpovídajících hodnotám vlastností ForegroundIndex a BackgroundIndex:

```
static TColor Barva[] = {
 clBlack, clMaroon, clGreen, clOlive, clNavy, clPurple,
 clTeal, clLtGray, clDkGray, clRed, clLime, clYellow,
 clBlue, clFuchsia, clAqua, clWhite,
};
```

Nás bude zajímat jen jedna barva a tu chceme vybírat pravým tlačítkem myši. Proto nastavíme vlastnost BackgroundEnable na false.

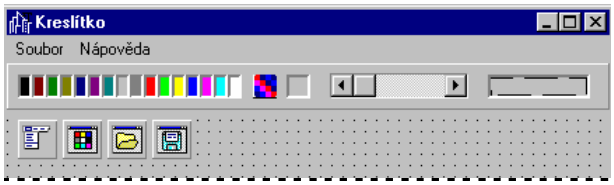
Dále upravíme tvar této mřížky. Chceme barevná pole uspořádat vodorovně v jedné řadě, a proto nastavíme vlastnost GridOrdering na go16x1. Nakonec upravíme velikost mřížky tak, aby zabírala méně než polovinu šířky okna (obr. 5.1).

Poznamenejme, že kliknutí na některém z políček této komponenty vyvolá událost OnChange.

Nakonec definujeme bublinovou nápovědu. Vlastnosti ShowHint přidělíme hodnotu true a vlastnosti Hint hodnotu Barva čáry.

Barevná mřížka ovšem nabízí jen základní barvy, a proto přidáme ještě tlačítko, které nám umožní vyvolat standardní dialogové okno pro volbu barvy. Vedle mřížky umístíme tlačítko s obrázkem (komponentu BitBtn). Jeho velikost upravíme tak, aby bylo přibližně stejné jako políčka barevné mřížky, a umístíme na ně bitovou mapu s nějakým výrazně barevným vzorem. (Připomeňme si, že bitovou mapu na tlačítku určuje vlastnost Glyph. Můžeme použít např. soubor volbar.bmp s adresáře KAP05\01 na doprovodném CD.) Vlastnost Caption vymažeme, takže tlačítko bude obsahovat pouze obrázek. Nakonec uložíme do vlastnosti Hint řetězec Jiná barva a vlastnost ShowHint nastavíme na true.

Vedle tlačítka umístíme další panel, který pojmenujeme pa_barva. Bude nám ukazovat aktuální barvu. Jeho velikost upravíme tak, aby byl stejně velký jako sousední tlačítko, vymažeme vlastnost Caption, vlastnost BevelOuter nastavíme na bvLowered (bude „nižší" než okolí) a vlastnosti BevelWidth dáme hodnotu 2 (tím zvýrazníme okraje). Do vlastnosti Hint uložíme řetězec Ukazuje nastavenou barvu a vlastnost ShowHint nastavíme na true.



Obr. 5.1 Vizuální návrh panelu nástrojů pro kreslítko

Nyní se naprogramuje nastavení barvy. Nejprve nastavíme v konstruktoru implicitní hodnoty:

```
__fastcall THlavniOkno::THlavniOkno(TComponent* Owner)
  : TForm(Owner), barva(clBlack), sila(1)
{}
```

Pak dvojklikem na barevnou mřížku barvy přimějeme prostředí, aby vytvořilo kostru handleru události OnChange, a doplníme tento kód:

```
void __fastcall THlavniOkno::barvyChange(TObject *Sender)
{
  static TColor Barva[] = {
    clBlack, clMaroon, clGreen, clOlive, clNavy, clPurple,
    clTeal, clLtGray, clDkGray, clRed, clLime, clYellow,
    clBlue, clFuchsia, clAqua, clWhite,
  };
  barva = Barva[barvy->ForegroundIndex];
  pa_barva -> Color =  barva;
}
```

Reakce na výběr barvy kliknutím na políčko barevné mřížky (soubor okno.cpp)

Zde nejprve uložíme do složky barva ve třídě THlavniOkno hodnotu barvy zjištěnou pomocí vlastnosti ForegroundIndex. Pak tuto barvu zobrazíme v okénku indikátoru barvy, tj. v panelu pa_barva.

Nyní vložíme do vizuálního návrhu okna komponentu ColorDialog, která zapouzdřuje standardní dialogové okno pro výběr barvy. Pojmenujeme ji dlg_zvolbarvu, ostatní vlastnosti ponecháme.

Naším dalším krokem bude vytvoření handleru, který reaguje na stisknutí tlačítka pro výběr některé z barev, které nejsou v barevné mřížce. V něm vyvoláme dialogové okno Barva, a pokud si uživatel vybere barvu a stiskne tlačítko OK, uložíme ji do složky barva a zobrazíme na panelu pa_barva:

```
void __fastcall THlavniOkno::tl_jinaClick(TObject *Sender)
{
  if(dlg_zvolbarvu->Execute())
  {
    barva = dlg_zvolbarvu->Color;
    pa_barva -> Color = barva;
  }
}
```

Výběr barvy pomocí dialogového okna Barva (soubor okno.cpp)

Poznamenejme, že uživatelem vybraná barva je obsažena ve vlastnosti Color dialogového okna.

Síla čáry

K zadávání síly čáry použijeme posuvník (komponentu ScrollBar) a zvolenou sílu čáry budeme zobrazovat v okénku na nástrojovém panelu vpravo (obr. 5.1).

Na panel nástrojů umístíme komponentu ScrollBar, kterou pojmenujeme posuv. Její vlastnost Kind nastavíme na sbHorizontal (chceme vodorovný, nikoli svislý posuvník) a upravíme její velikost. Vlastnosti Min a Max nastavíme na 1 a 100, takže bude-li táhlo úplně vlevo, bude mít vlastnost Position hodnotu 1, bude-li úplně vpravo, bude mít Position hodnotu 100. Vlastnostem SmallChange a LargeChange dáme hodnoty 1 a 10 a tím určíme, o kolik se táhlo posune při kliknutí na tlačítko na konci posuvníku a při kliknutí do prostoru mezi táhlem a tlačítkem na konci. Do vlastnosti Hint vložíme řetězec Nastavuje sílu čáry a vlastnost ShowHint nastavíme na true.

Při pohybu táhla vznikne událost OnChange.

Vedle posuvníku umístíme na nástrojový panel další komponentu Panel, kterou pojmenujeme pa_silapera; ponese komponentu, která bude znázorňovat zvolenou tloušťku čáry. Její velikost upravíme tak, jak ukazuje obr. 5.1 (poslední komponenta vpravo). Vlastnost Caption vymažeme, vlastnosti BevelOuter přiřadíme hodnotu bvLowered a vlastnosti BevelWidth hodnotu 2. Nakonec vložíme do vlastnosti Hint řetězec Ukazuje zvolenou sílu čáry a vlastnost ShowHint nastavíme na true.

Do tohoto panelu vložíme komponentu Image, pojmenujeme ji im_sila a její vlastnost Align nastavíme na alClient, takže se velikostí přesně přizpůsobí panelu pa_silapera. Na ní budeme zobrazovat zvolenou tloušťku čáry.

Pak napíšeme handler, který bude reagovat na posun táhla: Do složky sila uloží požadovanou tloušťku pera (tu zjistí z hodnoty vlastnosti Position) a uprostřed kreslicí oblasti komponenty im_sila nakreslí svislou černou čáru zvolené tloušťky.

```
void __fastcall THlavniOkno::posuvChange(TObject *Sender)
{
  sila = posuv->Position;          // Zjisti požadovanou tloušťku
  TCanvas * C = im_sila->Canvas;
  C -> FillRect(Rect(0,0,im_sila->Width, im_sila->Height));
  C->Pen->Width = sila;
  C->MoveTo(im_sila->Width/2, 0);
  C->LineTo(im_sila->Width/2, im_sila->Height);
}
```

Zobrazení tloušťky čáry

Poznámky

V prvním příkazu zjistíme požadovanou sílu čáry. Ve druhém řádku zavedeme pomocnou proměnnou, do které si uložíme ukazatel na plátno (canvas) komponenty im_sila. Tím ušetříme trochu psaní. V dalším příkazu vymažeme současný obsah této komponenty tím, že nakreslíme vyplněný obdélník v implicitní barvě (použijeme metodu FillRect()). Nakonec nastavíme sílu pera a nakreslíme svislou čáru v polovině šířky tohoto pole.

Implicitní nastavení na počátku

Nakonec se musíme postarat o počáteční nastavení barvy a tloušťky. Použijeme implicitní hodnoty (černá barva, tloušťka 1 bod) a nastavíme je v handleru, který se volá při vytvoření okna, tedy při události[30] OnCreate. Zavoláme v něm již hotové handlery posuvChange() a barvyChange():

```
void __fastcall THlavniOkno::FormCreate(TObject *Sender)
{
  posuvChange(Sender);
  barvyChange(Sender);
}
```

Později do tohoto handleru přidáme ještě volání metody, která inicializuje obrázek.

[30] To byl obvyklý postup v C++ Builderu 3.0. Ve verzi 4.0 došlo ke změně: Handler ošetřující událost OnCreate je volám konstruktorem předka, třídy TForm, nikoli konstruktorem třídy THlavniOkno. To znamená, že komponenty v okně nejsou v době volání tohoto handleru ještě vytvořeny. Proto je zpravidla rozumné spoléhat více na konstruktor třídy okna.

Třída hlavního okna

Do třídy THlavniOkno přidáme několik datových složek, které budou sdílet jednotlivé handlery. Především to bude ukazatel na instanci třídy Tobraz, která bude obsahovat naše dílo. Dále přidáme ukazatel na instanci třídy čára; do ní budeme ukládat jednotlivé body právě kreslené čáry.

O proměnných barva a sila, které budou obsahovat aktuální barvu čáry a její tloušťku, jsme hovořili v předchozích oddílech. K nim přibude ještě řetězcová proměnná jmeno, jež bude obsahovat jméno souboru, do kterého vytvořený obrázek uložíme, a logická proměnná dole, která bude obsahovat true, pokud bude stisknuto tlačítko myši.

```
class THlavniOkno : public TForm
{
__published:                    // IDE-managed Components
            // Handlery a ukazatele na komponenty vynecháme
private:                                // User declarations
   TColor barva;       // Aktuální barva
   unsigned sila;      // Aktuální síla pera
   Tobraz *obr;        // Kreslený obrázek
   Tcara * lajna;      // Aktuální čára
   bool dole;              // Zda je právě stisknuto tlačítko myši
   AnsiString jmeno;   // Jméno souboru
public:                                 // User declarations
   __fastcall THlavniOkno(TComponent* Owner);
};
```

Zkrácený výpis deklarace třídy hlavního okna programu Kreslítko (soubor okno.h)

Konstruktor této třídy bude mít za úkol tyto datové složky inicializovat:

```
__fastcall THlavniOkno::THlavniOkno(TComponent* Owner)
   : TForm(Owner), barva(clBlack), sila(1), dole(false),
     jmeno(implicitni_nazev)
{}
```

Nový soubor

Nyní naprogramujeme odezvu na příkaz Soubor | Nový. Handler, který bude výběr této položky ošetřovat, vymaže předchozí obrázek (tj. data uložená v instanci typu Tobraz) a vytvoří nový, prázdný. Zároveň nastaví implicitní jméno souboru.

Vedle toho nastaví i rozměry kresleného obrázku. Komponenta Image má, jak už víme, vlastnost Picture, a ta má vlastnost Bitmap. Tato bitová mapa má na počátku rozměry 0 × 0 pixelů, takže se nezobrazuje — celý obrázek je po spuštění programu šedý, má barvu podkladu. Tím, že nastavíme jeho rozměry, způsobíme, že se vyplní bílou barvou (implicitním štětcem).

Nakonec smažeme obrázek, který komponenta Image obsahuje. K tomu použijeme metodu TCanvas::FillRect().

```
const char * implicitni_nazev = "kresba.kre";

void __fastcall THlavniOkno::mn_novyClick(TObject *Sender)
{
   delete obr;                                  // Zruš předchozí obrázek
   obr = new Tobraz;                        // Vytvoří nový, prázdný
   jmeno = implicitni_nazev;            // Nastaví implicitní jméno
   int x = pa_obr->Width;               // Zjistí rozměry komponenty Image
   int y = pa_obr -> Height;
   im_kresba -> Picture-> Bitmap->Height = y;          // Nastaví rozměry
   im_kresba -> Picture-> Bitmap->Width = x;           // bitové mapy
   im_kresba -> Canvas -> FillRect(Rect(0,0,x,y));// a vymaže obrázek
}
```

Handler, který zpracovává příkaz Soubor | Nový (soubor okno.cpp)

Metodu THlavniOkno::mn_novyClick() zavoláme také v handleru FormCreate(), který zpracovává událost OnCreate hlavního okna. Tím zajistíme, že hned po spuštění programu bude připravena bitová mapa správných rozměrů.

```
void __fastcall THlavniOkno::FormCreate(TObject *Sender)
{
   posuvChange(Sender);
   barvyChange(Sender);
   mn_novyClick(Sender);
```

```
}
```

Kreslení

Nyní naprogramujeme vlastní kreslení. Přitom se budeme zabývat třemi událostmi: Kreslení čáry začne při OnMouseDown; při ní nastavíme příznak, že bylo stisknuto tlačítko myši, uvolníme z paměti aktuální čáru, vytvoříme novou se zvolenou barvou a tloušťkou pera a uložíme do ní první bod (ten, ve kterém jsme stiskli tlačítko myši). Zároveň se připravíme k nakreslení první z úseček tvořících čáru, tj. přejdeme do počátečního bodu čáry.

Vlastní kresba bude vznikat při událostech OnMouseMove. Zde nejprve zjistíme, zda je nastaven příznak stisknutí tlačítka. (Není-li stisknuto tlačítko, pohyb myši nás nezajímá.) Pokud ano, připojíme k aktuální čáře nový bod a do obrázku nakreslíme novou úsečku.

Při události OnMouseUp kreslení aktuální čáry skončí. I zde samozřejmě nejprve zjistíme, zda bylo stisknuto tlačítko, a pokud ano, připojíme k aktuální čáře poslední bod, nakreslíme poslední úsečku, právě dokončenou čáru připojíme k obrázku a zrušíme příznak, že bylo stisknuto tlačítko.

```
void __fastcall THlavniOkno::FormMouseDown(TObject *Sender,
                                    TMouseButton Button, TShiftState Shift, int X, int Y)
{
  dole = true;                          // Nastav příznak, že se kreslí
  delete lajna;                             // Zruš předchozí čáru
  lajna = new Tcara(barva, sila);  // a vytvoř novou
  lajna->pridej(Tbod(X, Y));            // Vlož do ní počáteční bod
  im_kresba->Canvas->Pen->Color = barva;        // Nastav barvu a
  im_kresba->Canvas->Pen->Width = sila;  // sílu pera a přejdi
  im_kresba->Canvas->MoveTo(X,Y);               // do počátečního bodu
}
void __fastcall THlavniOkno::FormMouseMove(TObject *Sender,
                                                                    TShiftState Shift, int X, int Y)
{
  if(dole)                                          // Jen pokud je stisknuto tlačítko
  {
   lajna->pridej(Tbod(X, Y));           // Přidej nový bod do čáry
   im_kresba->Canvas->LineTo(X,Y); // Nakresli novou úsečku
  }
}
void __fastcall THlavniOkno::FormMouseUp(TObject *Sender,
                                TMouseButton Button, TShiftState Shift, int X, int Y)
{
  if(dole)                                          // Jen pokud je stisknuto tlačítko
  {
    dole = false;                           // Zruš příznak stisknutého tlačítka
    lajna->pridej(Tbod(X, Y));      // Přidej poslední bod
    im_kresba->Canvas->LineTo(X,Y);// Nakresli novou úsečku
    obr->pridej(lajna);             // Ulož novou čáru
  }
}
```

Kreslení čáry (soubor okno.cpp)

Poznámky

- ✧ Instance *obr třídy Tobraz není pro vlastní kreslení potřebná. Použijeme ji, až budeme chtít náš obrázek uložit.
- ✧ Handlery ošetřující události OnMouse... mají mj. parametr Shift typu TShiftState, který obsahuje také informace o tom, které tlačítko myši bylo stisknuto. Pro nás ale nemají význam: Události OnMouseMove a OnMouseUp nás zajímají jen v případě, že uživatel stiskl tlačítko myši nad klientskou oblastí okna programu. Proto jsme použili proměnnou dole.

Uložení souboru

Nejprve se podíváme, zda byl náš soubor už alespoň jednou uložen, tj. zda byl pojmenován. To zjistíme, porovnáme-li obsah proměnné jmeno s implicitním jménem, které je uloženo v globální proměnné *implicitni_nazev. Pokud už byl uložen, určíme podle přípony požadovaný typ souboru a

zavoláme metodu Tobrazek::uloz() nebo TPicture::SaveToFile(), jinak zavoláme handler, který má na starosti příkaz Ulož jako....

```
void __fastcall THlavniOkno::mn_ulozClick(TObject *Sender)
{
if(jmeno == implicitni_nazev)              // Byl už soubor uložen?
      mn_ulozjakoClick(Sender);            // Pokud ne, ulož jako...
  else                                                    // Jinak urči podle přípony
   {                                                      // způsob uložení
    if(ExtractFileExt(jmeno) == ".kre")
      obr->uloz(jmeno);
    else if(ExtractFileExt(jmeno) == ".bmp")
      im_kresba -> Picture -> SaveToFile(jmeno);
   }
}
```

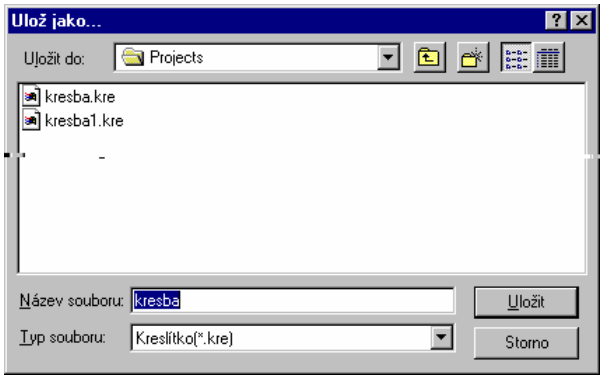
Uložení obrázku do souboru (soubor okno.cpp)

Uložení pod jiným názvem

Tento příkaz umožní uživateli uložit soubor pod jiným názvem nebo i v jiném formátu. (Podle požadavků, které jsme si na počátku položili, to může být vedle vlastního formátu kreslítka také bitová mapa.) Pro určení jména, cesty a formátu použijeme standardní dialogové okno pro uložení souboru, které v našem programu představuje komponenta dlg_ulozjako typu TSaveDialog (obr. 5.2).

Do vlastnosti FileName uložíme aktuální jméno souboru, které je uloženo v proměnné jmeno; odstraníme z něj však příponu, aby nebylo nutno ji měnit při změně typu souboru. K tomu použijeme metodu AnsiString::Delete().

Pokud uživatel stiskne v tomto okně tlačítko Uložit, přečteme si z vlastnosti FileName jméno souboru. Pokud soubor tohoto jména už existuje, zeptá se, zda jej smí přepsat. Pak ze jména souboru vyjme příponu, podle ní určí typ souboru a použije odpovídající metodu. Nové jméno souboru uloží do složky jmeno.



Obr. 5.2 Standardní dialogové okno pro uložení souboru pod jiným názvem

```
void __fastcall THlavniOkno::mn_ulozjakoClick(TObject *Sender)
{
  dlg_ulozjako -> FileName = jmeno.Delete(jmeno.LastDelimiter("."),4);
  if(dlg_ulozjako->Execute())
  {
    jmeno = dlg_ulozjako->FileName;              // Zjisti jméno souboru
    if(FileExists(jmeno))                  // Zjisti, zda už existuje
    {                                                       // a pokud ano, zeptej se,
     if(Application->MessageBox(("Soubor "+jmeno+" existuje.\n"
       "Mám ho přepsat?").c_str(),"Upozornění",
                  MB_YESNO|MB_ICONQUESTION)==IDNO)
      return;                                       // zda ho chce přepsat
    }
    if(ExtractFileExt(jmeno) == ".kre")        // Podle přípony urči
      obr->uloz(jmeno);                        // způsob uložení
    else if(ExtractFileExt(jmeno) == ".bmp")
      im_kresba -> Picture -> SaveToFile(jmeno);
  }
}
```

Uložení souboru pod jiným jménem (soubor okno.cpp)

Poznámky

- ✧ Kód pro uložení obrázku do souboru je zde stejný jako v THlavniOkno::mn_ulozClick(). Bylo by tedy vhodné uložit jej do samostatné funkce.
- ✧ Před vlastním zápisem do souboru testujeme, zda soubor tohoto jména náhodou neexistuje, pomocí funkce FileExists(). C++ Builder ve skutečnosti nabízí jednodušší možnost založenou na vlastnosti ofOverwritePrompt, která je součástí vlastnosti Options. Přiřadíme-li jí hodnotu true., bude toto okno kontrolovat, zda daný soubor náhodou neexistuje, a v případě potřeby nás upozorní.

Změna typu ukládaného souboru

Dialogové okno Ulož jako zobrazuje soubory typů určených implicitní příponou — tedy vlastností DefaultExt. Jestliže změníme typ ukládaného souboru, měl by se změnit i typ zobrazovaných souborů. Změna typu souboru způsobí událost OnTypeChange; ošetříme ji a v handleru změníme implicitní příponu.

```
void __fastcall THlavniOkno::dlg_ulozjakoTypeChange(TObject *Sender)
{
  static AnsiString ext[] = {".kre", ".bmp"};
  dlg_ulozjako -> DefaultExt =  ext[dlg_ulozjako -> FilterIndex-1];
}
```

Otevření souboru

Jméno a cestu souboru určíme pomocí standardního dialogového okna pro otevírání souboru, které je zapouzdřeno v komponentě dlg_otevri typu TOpenDialog. Pokud uživatel stiskne v tomto okně tlačítko Otevřít, zjistíme z vlastnosti FileName jméno souboru. Pak vymažeme aktuální obrázek (data uložení v obr* i obsah okna), vytvoříme nový prázdný, pomocí metody Tobrazek::precti() přečteme soubor a pomocí Tobrazek::nakresli() ho nakreslíme.

```
void __fastcall THlavniOkno::mn_otevriClick(TObject *Sender)
{
  if(dlg_otevri->Execute())
  {
   AnsiString jmeno = dlg_otevri->FileName;          // Zjisti jméno souboru
   delete obr;                                                  // Zruš starý a vytvoř
   obr = new Tobraz;                                          // nový obrázek
   im_kresba -> Canvas -> FillRect(                    // Vymaž okno
                          Rect(0,0,im_kresba->Width, im_kresba->Height));
   obr-> precti(jmeno);                                  // Přečti soubor
   obr->nakresli(im_kresba->Canvas);                  // a nakresli obrázek
  }
}
```

Přečtení obrázku ze souboru

O programu

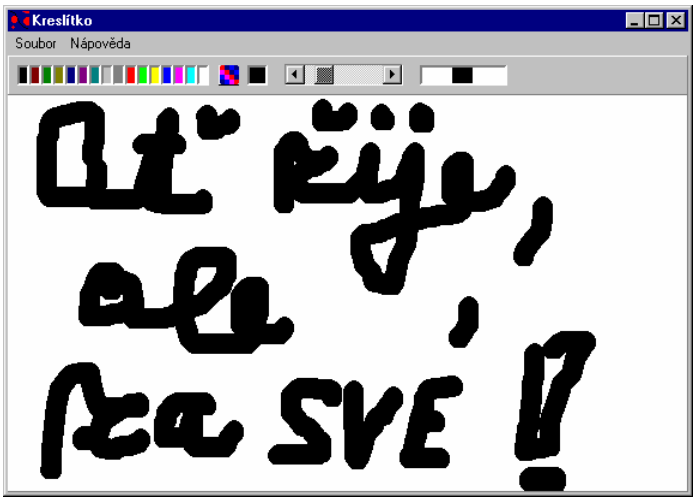
Zbývá vytvořit dialogové okno O programu a ošetřit odpovídající položku z nabídky. To jistě zvládnete sami.

Poznámky

Výsledek ukazuje obrázek 5.3; zdrojový program kreslítka verze 1.0 najdete v adresáři KAP05\01 na doprovodném CD.

Program v této podobě sice funguje, ale ke slušnému softwaru má ještě hodně daleko. Jeho nejnápadnější vadou je, že nás bez reptání poslechne, jestliže mu přikážeme smazat rozdělaný obrázek. Bylo by vhodné, kdyby se v situacích, kdy můžeme přijít o výsledek své práce, nejprve zeptal, zda to myslíme vážně. Zkuste ho tímto způsobem upravit, není to těžké. (Výsledek najdete na doprovodném CD v adresáři KAP05\02)

Dále se můžete upravit formát ukládaného souboru tak, aby obsahoval i rozměry obrázku v pixelech. Program by měl umožňovat vrátit poslední editační akci, tj. smazat poslední čáru — a lze vymyslet ještě mnoho dalších vylepšení. Pokuste se o ně.



Obr. 5.3 A můžeme si kreslit...

5.3 Tisk

Zbývá poslední důležitá věc, kterou náš program neumí, a to tisk obrázku.

Asi nejjednodušší možnost, kterou nám C++ Builder nabízí, je použití zděděné metody TForm::Print(), která vytiskne obsah okna jako bitovou mapu. Není to ovšem dobrá možnost, neboť

- ✧ vytiskne jen to, co je v aktuálním okně,
- ✧ vytiskne nejen to, co jsme nakreslili, ale i všechny řídicí prvky, které v okně jsou — v našem případě i nástrojový panel,
- ✧ vytiskne i část rámečku okna.

Rozumnější je využít služeb globálního objektu typu TPrinter, který představuje implicitní (v terminologii českých Windows *výchozí*) tiskárnu vašeho počítače.

Typ TPrinter je definován v hlavičkovém souboru printers.hpp, který musíme do svého programu vložit. Ukazatel na globální tiskárnu získáme voláním funkce Printer().

Tisk zahájíme voláním metody BeginDoc() a ukončíme voláním EndDoc(). Na novou stránku přejdeme voláním metody NewPage(). Vlastní tisk je stejný jako kreslení do okna — využijeme vlastnost Canvas. Pro vytisknutí obrázku tedy můžeme využít metodu Tobraz::nakresli(), které předáme jako parametr plátno tiskárny.

Pokud nám tedy stačí tisk na implicitní tiskárnu, můžeme postupovat takto:

- ✧ Do nabídky Soubor přidáme položku Tisk.... Odpovídající komponentu typu TMenuItem uložíme pod jménem mn_tisk.
- ✧ Napíšeme handler, který bude reagovat na vybrání tohoto příkazu:

```
#include <printers.hpp>

void __fastcall THlavniOkno::mn_tiskClick(TObject *Sender)
{
  TPrinter * Tiskarna = Printer();
  Tiskarna -> BeginDoc();
  obr-> nakresli(Tiskarna -> Canvas);
  Tiskarna -> EndDoc();
}
```

Tento program funguje, ale přece jen není dokonalý — chybí zde např. možnost předepsat počet kopií, tisknout do souboru atd. K tomu bychom si mohli vytvořit dialogové okno, které zjistí od uživatele potřebné údaje a použije je k nastavení vlastnosti tiskárny Copies, která určuje požadovaný počet kopií, Orientation, která určuje orientaci papíru (na výšku nebo na šířku) a další. Vlastnost Printers

obsahuje seznam všech nainstalovaných tiskáren ve Windows a vlastnost PrinterIndex říká, která tiskárna z tohoto seznamu je aktuální.

Vlastnosti Printing a Aborted ukazují, zda tiskárna právě tiskne nebo zda byl tisk na ní zrušen.

Ve skutečnosti se však o většinu vlastností nemusíme vůbec starat, stačí použít standardní komponenty PrintDialog, kterou najdeme na paletě Dialogs. Veškerá nastavení v tomto dialogu se automaticky přenášejí na aktuální tiskárnu.

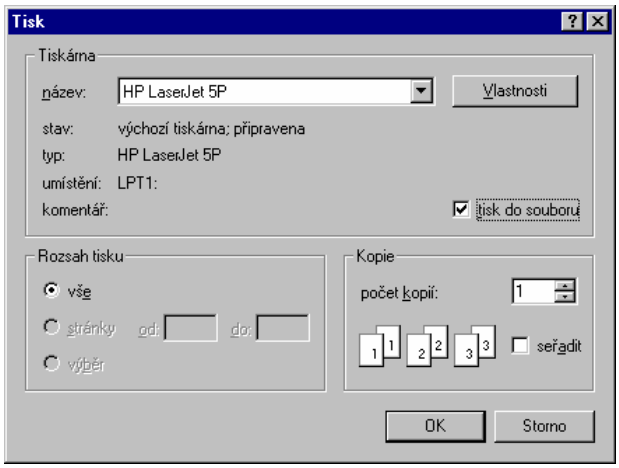
Vložíme tedy do vizuálního návrhu programu tuto komponentu a pojmenujeme ji dlg_tisk. Mezi jejími vlastnostmi v inspektoru objektů vyhledáme možnost Options, která určuje vzhled a chování tohoto dialogového okna, a kliknutím na křížek před ní zobrazíme jednotlivé dílčí volby. Najdeme vlastnost poPrintToFile a nastavíme ji na true — tím povolíme zobrazování zaškrtávacího pole Tisk do souboru. V ostatních případech ponecháme implicitní hodnoty.

Nakonec upravíme handler, který bude reagovat na vybrání příkazu tisk, do následující podoby:

```
void __fastcall THlavniOkno::mn_tiskClick(TObject *Sender)
{
  dlg_tisk -> Copies = 1;                  // Implicitní počet kopií
  static TPrinter * Tiskarna = Printer();  // Objekt tiskárny
  if(dlg_tisk->Execute())                  // Pokud uživatel stiskl OK...
  {
    Tiskarna -> BeginDoc();
    obr-> nakresli(Tiskarna -> Canvas);
    Tiskarna -> EndDoc();
  }
}
```

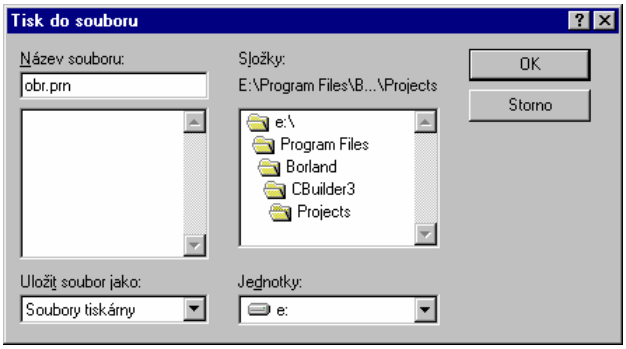
Handler, který se stará o tisk

Zadáme-li nyní příkaz Soubor | Tisk, objeví se známé dialogové okno (obr. 5.4). Nastavíme-li zde např. požadavek na tisk 3 kopií, vytisknou se opravdu 3 kopie; o to se postará komponenta PrintDialog.



Obr. 5.4 Dialogové okno pro tisk souboru

Podobně zařídí vše potřebné i v případě, že zaškrtneme políčko tisk do souboru. Jestliže pak stiskneme OK, objeví se automaticky dialogové okno, které zjistí jméno souboru (obr. 5.5).



Obr. 5.5 Standardní okno pro zjištění jména souboru, do kterého se bude tisknout

Tím bychom mohli skončit. (Program v této podobě najdete na doprovodném CD v adresáři KAP05\03.) Program tiskne, používá k tomu standardní dialogová okna, umí tisknout i do souboru... Při pohledu na vytištěný dokument ale zjistíme, že zde teprve začíná skutečná práce. Takto vytištěný obrázek má totiž několik nedostatků, z nichž asi nejzávažnější je, že velikost tištěného obrázku závisí na rozlišení tiskárny a na rozlišení obrazovky, ve kterém byla vytvořena předloha. Vedle toho náš program neumožňuje vybrat si jednu z nainstalovaných tiskáren v případě, že jich je k disposici více. (K tomu bychom mohli použít standardní dialogové okno zapouzdřené v komponentě PrinterSetupDialog.) Nezbývá tedy než se pustit do další práce a začít program vylepšovat. Pusťte se do toho.

6. Pokusy s fraktály

V této kapitole napíšeme program, který bude počítat a zobrazovat některý z fraktálů, např. Mandelbrotovu množinu. Pokud nemáte tušení, o co jde, neděste se, nepůjde o nic složitého; fraktály nám poslouží především jako příklad časově náročného výpočtu a problémů, na které přitom můžeme narazit. Přitom se také seznámíme s multithreadingem a poznáme několik nových komponent.

6.1 Verze 1.0

Než se pustíme do práce, ujasníme si, co vlastně chceme. Fraktál je jakási množina bodů v rovině a náš program by ji měl nakreslit. Měl by např. černě zobrazit body, které do této množiny patří, a červeně body, které do ní nepatří. To znamená, že pro každý z bodů (pixelů) obrázku musí vypočítat jeho barvu.

Přitom bude nutné, abychom mohli zadat oblast v rovině, kterou chceme „zmapovat“. Tu nebudeme zadávat v souřadnicích okna, tedy v pixelech, ale v souřadnicích obvyklých pro práci s fraktály — tedy v reálných číslech.

Velikost výsledného obrázku se bude v první verzi řídit velikostí okna a vytvořené obrázky budeme chtít ukládat do bitmapových souborů.

Fraktál

Už jsme si řekli, že fraktál je pro nás jakási množina bodů v rovině. Nebudeme se zabývat přesnou definicí, řekneme si pouze, že fraktály jsou v jakémsi smyslu nekonečně složité a jejich jednotlivé části jsou podobné sobě navzájem i celé množině. Už to může být dobrý důvod, proč psát program, který je zobrazuje.[31]

Nepochybně nejznámějším fraktálem je Mandelbrotova množina. To je množina všech bodů z v komplexní rovině, pro které konverguje posloupnost c_n definovaná vztahy

$$c_0 = z,$$
$$c_{n+1} = c_n * c_n + z. \qquad (1)$$

Jiná zajímavá množina je např. množina bodů z, pro které konverguje posloupnost

$$c_0 = z,$$
$$c_{n+1} = c_n * (c_n - 1). \qquad (2)$$

Jak ale zjišťovat v programu konvergenci takovéto posloupnosti? Použijeme jednoduchý trik: Vezmeme nějaké dostatečně velké N a pomocí vztahů (1) vypočteme postupně c_0, c_1, c_2, … , c_N. Bude-li reálná nebo imaginární část čísla c_N v absolutní hodnotě větší než dvě, prohlásíme, že tato posloupnost diverguje, jinak budeme tvrdit, že konverguje. (Matematikům při tom možná vstávají hrůzou vlasy na hlavě, ale ono to kupodivu v našem případě bude docela dobře fungovat.)

Výpočet

V předchozím odstavci jsme viděli, že fraktály, které nás zajímají, jsou definovány pomocí komplexních čísel. To nepředstavuje problém: Komplexní číslo je vlastně dvojice reálných čísel, a tak si můžeme říci, že bodu o souřadnicích (x, y) v rovině obrázku bude odpovídat komplexní číslo

$$z = x + iy,$$

kde i je imaginární jednotka. Komplexní čísla se v technicky orientovaných programech objevují poměrně často, a proto obsahuje standardní knihovna jazyka C++ šablonu `complex<T>`, jejíž instance jsou komplexní čísla vytvořená z dvojic čísel typu `T`. Najdeme ji v hlavičkovém souboru `complex` a jako

[31] Podrobnější výklad o fraktálech a mnoha dalších věcech, které s nimi přímo i nepřímo souvisí, můžete najít v knize [7], která je určena laikům. Tam najdete i případné další odkazy na matematickou literaturu, která se touto problematikou zabývá.

všechny součásti standardní knihovny leží v prostoru jmen std. Reálnou a imaginární část komplexního čísla zjistíme pomocí metod real() a imag().

Funkci, která bude počítat barvu jednotlivých bodů, uložíme do samostatného souboru vypoc.cpp. Bude mít dva parametry typu double určující souřadnice bodu v rovině a jeden parametr typu int určující počet členů posloupnosti (1), které budeme při výpočtu zkoumat, a bude vracet hodnotu typu TColor:

```
#ifndef vypocH
 #define vypocH
TColor Vypocet(double x, double y, int n);
#endif
```

Prototyp funkce Vypocet (soubor vypoc.h)

Pro zpřehlednění zápisu si definujeme pomocnou funkci diverguje(c), která bude testovat, zda má některá ze složek komplexního čísla c absolutní hodnotu větší než 2. Samotná funkce Vypocet() vytvoří z předaných parametrů komplexní číslo a bude v cyklu počítat členy posloupnosti (1). Pokud po vypočtení některého z členů zjistí, že posloupnost diverguje, výpočet skončí a funkce vrátí clRed (červená barva, bod nepatří do množiny). Pokud proběhne zadaný počet opakování, znamená to, že posloupnost konverguje, takže bod patří do množiny a funkce Vypocet vrátí clBlack (černá, bod patří do množiny).

```
#include <vcl.h>                        // Kv li typu TColor
#include "okno1.h"
#include "vypoc.h"
#include <complex>                      // Komplexní čísla

using namespace std;

typedef complex<double> cplx;           // Abychom nemuseli tolik ps t...

inline bool diverguje(cplx c)           // Test, zda m  n kter  slo ka c
{                                               // absolutní hodnotu větší než 2
 if(c.real() > 2 || c.real() < -2 || c.imag() > 2 || c.imag() < -2) return true;
 else return false;
}

TColor Vypocet(double x, double y, int n)    // Vlastn  v po et barvy
{
  complex<double> z(x,y), d(z);
  for(int i = 0; i < n; i++)
  {
    z = z*z+d;                          // Výpočet nového členu posloupnosti
    if(diverguje(z)) return clRed;      // Pokud diverguje, skonči a vrať
  }                                             // červenou, jinak bod leží v množině,
  return clBlack;                       // takže vrať černou
}
```

Funkce pro výpočet barvy bodu obrázku (soubor vypoc.cpp)

Hlavní okno

Začneme nový projekt, který uložíme pod jménem Fra1.cpp. Hlavní okno pojmenujeme jako obvykle HlavniOkno a odpovídající modul uložíme pod názvem okno1.

Nejprve vytvoříme nabídku. Do vizuálního návrhu okna vložíme komponentu MainMenu, kterou pojmenujeme HlavniMenu. V návrháři menu vytvoříme tyto položky:

- Nabídka Soubor bude mít podřízené položky Nový..., Ulož... a Konec. Odpovídající komponenty pojmenujeme mn_movy, mn_uloz a mn_konec. Pro položku Ulož... definujeme tradiční klávesovou zkratku CTRL+S: V inspektoru objektů napíšeme jako hodnotu vlastnosti ShortCut řetězec Ctrl+S.
- Nabídka Obrázek bude mít podřízené položky Nastavení... a Výpočet. Odpovídající komponenty pojmenujeme mn_nastaveni a mn_vypocet.
- Nabídka Nápověda bude obsahovat obvyklou položku O programu.... Odpovídající komponentu pojmenujeme mn_oprogramu.

Význam jednotlivých položek je zřejmý z jejich názvů. Jen zdůrazníme, že výpočet a kreslení bodů obrázku nezačne hned při spuštění programu, ale až po zadání příkazu Obrázek | Výpočet.

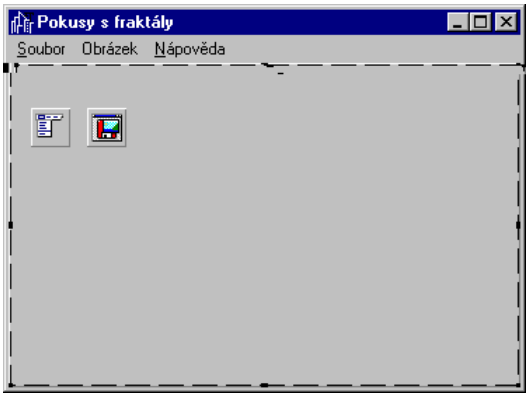
Dále přidáme do hlavního okna komponentu SavePictureDilaog, tj. dialogové okno pro uložení obrázku. Pojmenujeme ji dlg_ulozjako, do její vlastnosti Filtr uložíme řetězec "Bitové mapy (*.bmp)|*.bmp", do

vlastnosti DefaultExt řetězec "bmp" a do vlastnosti Title řetězec "Ulož jako...". Okno tedy bude zobrazovat pouze soubory s příponou .bmp a pokud u jména souboru tuto příponu neuvedeme, doplní ji.

Pak klikneme v inspektoru objektů na křížek vedle vlastnosti Options a tím zobrazíme jednotlivé dílčí vlastnosti, ze kterých se skládá. Vyhledáme vlastnost ofOverwritePrompt a přiřadíme jí hodnotu true; tím zabezpečíme, že pokud soubor zadaného jména existuje, zeptá se nás program, zda jej smí přepsat.

Nakonec připravíme kreslicí plochu. Do vizuálního návrhu okna vložíme komponentu Panel, pojmenujeme ji pa_obr, vlastnosti Caption přiřadíme prázdný řetězec a vlastnosti Align hodnotu alClient, takže vyplní celou klientskou oblast okna. Na tento panel položíme komponentu Image, kterou pojmenujeme im_obr. Její vlastnosti Align přiřadíme hodnotu alClient, takže vyplní celý panel. Do této komponenty budeme kreslit fraktály.

Vizuální návrh hlavního okna ukazuje obr. 6.1.



Obr. 6.1 Vizuální návrh okna pro kreslení fraktálů

Do definice třídy okna dále přidáme soukromé datové složky, ve kterých si budeme pamatovat rozměry okna v pixelech, „skutečné“ souřadnice levého dolního a pravého horního rohu obdélníka (intervalu) z komplexní roviny, ve kterém náš fraktál leží, počet iterací (tj. maximální členů posloupnosti (1), které budeme zkoumat), a také příznak, zda jsme vytvořený obrázek uložili. Komentáře v následujícím výpisu vysvětlují jejich význam. Pro většinu z těchto složek definujeme také veřejné přístupové funkce.

Definice třídy THlavniOkno bude po těchto úpravách vypadat např. takto (vynecháme ukazatele na komponenty):

```
class THlavniOkno : public TForm
{
__published:                                    // IDE-managed Components
        // Ukazatele na komponety vynecháme...
private:                                                // User declarations
  int pocetIteraci;
  double XLH, YLH, XPD, YPD;      // Souřadnice levého horního a pravého dolního
                                                        // rohu zkoumaného intervalu v rovině X-Y
  int sirka, vyska;                 // Rozměry bitové mapy v pixelech
  double ax, ay;                    // Konstanty pro přepocet souřadnic na pixely
  bool ulozeno;                             // Byl už obrázek uložen?
  void VypoctiKonstanty();          // Výpočet ax a ay
  int DotazChcesUlozit();       // Obrázek nebyl uložen. Chceš ho uložit?
  void Nastaveni();                 // Nastavení některých hodnot
public:                                                 // User declarations
  __fastcall THlavniOkno(TComponent* Owner);
  int PocetIteraci() {return pocetIteraci;}  // Přístupové funkce
  int Sirka() { return sirka; }
  int Vyska() { return vyska; }
  double xlh() { return XLH; }
  double xpd() { return XPD; }
  double ylh() { return YLH; }
  double ypd() { return YPD; }
  double Ax() { return ax; }
  double Ay() { return ay; }
  double X(int x) { return ax*x+XLH; }          // Přepočet souřadnic v okně (pixelů)
  double Y(int y) { return ay*y+YLH; }          // na souřadnice v rovině
};
```

Zkrácená deklarace třídy THlavniOkno (soubor okno1.cpp)

Funkce DotazChcesUlozit() vyvolá dialogové okno s upozorněním, že obrázek nebyl uložen, a zeptá se uživatele, zda si ho přeje uložit. Uživatel stiskne jedno z tlačítek Ano, Ne a Storno a funkce vrátí odpovídající výsledek. Její definice v souboru okno1.cpp je jednoduchá:

```
int THlavniOkno::DotazChcesUlozit()
{
 return
  Application -> MessageBox(    "Obrázek nebyl uložen.\nChcete ho uložit?",
                                "Upozornění", MB_YESNOCANCEL);
}
```

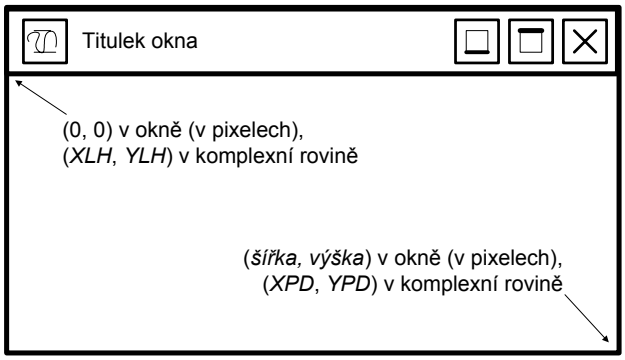
Přepočet souřadnic

Postup kreslení bude jednoduchý: Vezmeme jeden bod v okně (pixel) po druhém a určíme jeho barvu, tedy zjistíme, zda patří do zkoumané množiny. Pak ho ve vypočtené barvě nakreslíme.

Funkce Vypocet() však očekává souřadnice v komplexní rovině. Proto musíme okenní souřadnice vyjádřené v pixelech na souřadnice v rovině přepočítat. Snadno zjistíme (obr. 6.2), že přepočtové vztahy mají tvar

$$x' = a_x . x + b_x = \frac{XPD - XLH}{sirka} x + XLH, \quad y' = a_x . y + b_y = \frac{YPD - YLH}{vyska} + YLH,$$

kde x, y jsou souřadnice v pixelech a x', y' jsou souřadnice v komplexní rovině.



Obr. 6.2 Vztahy souřadnic v okně

Konstanty a_x a a_y si uložíme do složek ax a ay hlavního okna. Jejich výpočet bude mít na starosti funkce

```
void THlavniOkno::VypoctiKonstanty()
{
 ax = (XPD-XLH)/sirka;
 ay = (YPD-YLH)/vyska;
}
```

kterou budeme volat na počátku při vytvoření okna (tj. při události OnCreate), při změně souřadnic v komplexní rovině (tj. v handleru, který bude ošetřovat příkaz nabídky Nastavení...) a před začátkem kreslení nového obrázku. Na druhé straně ji nebudeme volat při změně rozměrů okna, neboť k té může dojít i v průběhu kreslení, a tím bychom změnili parametry uprostřed výpočtu obrázku.

Pro přepočet pixelů na souřadnice budou sloužit funkce X() a Y(), jejichž definice vidíte na výpisu třídy THlavniOkno.

Inicializace

Konstruktor třídy THlavniOkno bude mít jako obvykle na starosti inicializaci datových složek uložených v instanci hlavního okna. Abychom nemuseli při každém spuštění programu zadávat souřadnice zkoumané části roviny, počet iterací a další údaje, deklarujeme jejich počáteční hodnoty jako konstanty. Vytvoříme proto zvláštní soubor, který pojmenujeme init.h.

```
// Po   te n   po et iterac
const int InitPocetIteraci = 500;
```

```
// Po  te n  sou adnice levého horn ho (LH) a pravého doln ho (PD)
// rohu obrázku v rovině X-Y
const double InitXLH = -2;
const double InitYLH = 1.5;
const double InitXPD = 1;
const double InitYPD = -1.5;
#endif
```

Počáteční hodnoty v souboru init.h

Toto nastavení říká, že pokud uživatel nic nezmění, bude program zkoumat obdélník, v němž souřadnice *x* bude v rozmezí (-2; 1,5) a souřadnice *y* v rozmezí (-1,5; 1,5) a přitom vypočte maximálně 500 členů posloupnosti (1).

Konstruktor třídy THlavniOkno bude mít tvar

```
__fastcall THlavniOkno::THlavniOkno(TComponent* Owner)
  : TForm(Owner), pocetIteraci(InitPocetIteraci),
  XLH(InitXLH), YLH(InitYLH), XPD(InitXPD), YPD(InitYPD),
  sirka(im_obr->Width), vyska(im_obr->Height), ulozeno(true)
{
  Nastaveni();          // Nastavení šířky a výšky - viz dále
}
```

Konstruktor třídy hlavního okna (soubor okno1.cpp)

K inicializaci složek sirka a vyska můžeme použít vlastností komponenty im_obr, a to im_obr->Width a im_obr->Width. Podle pravidel jazyka C++ se totiž nejprve volá konstruktor předka, třídy TForm, a ten se postará o vytvoření všech komponent v okně. Teprve pak se inicializují datové složky deklarované v odvozené třídě. To nám zaručuje, že v době inicializace složek sirka a vyska již komponenta im_obr existuje.

Proměnné ulozeno přiřadíme na počátku hodnotu true: po spuštění neobsahuje program žádný obrázek, a proto není třeba uživatele obtěžovat dotazy na uložení.

Po vytvoření okna musíme nastavit rozměry bitové mapy, do které budeme kreslit, a vypočítat konstanty ax a ay pro přepočet pixelů na souřadnice v komplexní rovině. V předchozí verzi C++ Builderu bychom to mohli svěřit handleru pro událost OnCreate, avšak v současné verzi nikoli[32]. Napíšeme tedy pomocnou funkci Nastaveni(), kterou tím pověříme, a zavoláme ji z těla konstruktoru:

```
void Nastaveni()
{
  im_obr -> Picture -> Bitmap -> Height = vyska;
  im_obr -> Picture -> Bitmap -> Width = sirka;
  VypoctiKonstanty();
}
```

Nastavení rozměrů bitové mapy (soubor okno1.cpp)

Výpočet

Výpočet se rozeběhne po zadání příkazu Obrázek | Výpočet. Odpovídající handler nejprve nastaví příznak, že obrázek dosud nebyl uložen, a pak v cyklu projde všechny body (pixely) obrázku a vypočte jejich barvu. Přitom využije vlastnost Pixels plátna (canvasu); připomeňme si, že to je pole odpovídající jednotlivým grafickým bodům kreslicí plochy a hodnotou složek tohoto pole je jeho barva. Obrázek nakreslíme tím, že jednotlivým bodům plátna přiřadíme barvu.

Indexy prvků pole Pixels představují „okenní“ souřadnice. Před výpočtem je musíme přepočítat na souřadnice v komplexní rovině; k tomu poslouží funkce X() a Y().

```
void __fastcall THlavniOkno::mn_vypocetClick(TObject *Sender)
{
  ulozeno = false;                              // Obrázek ještě nebyl uložen
  for(int ix = 0; ix < sirka; ix++)
   for(int iy = 0; iy < vyska; iy++)
    im_obr -> Canvas -> Pixels[ix][iy] = Vypocet(X(ix), Y(iy), pocetIteraci);
}
```

[32] V současné verzi, tj. v C++ Builderu 4, se o volání handleru ošetřujícího událost OnCreate stará konstruktor předka, tedy třídy TForm. To znamená, že v době volání tohoto handleru ještě neproběhly inicializace ostatních složek odvozené třídy THlavniOkno a neproběhlo tělo konstruktoru. V předchozí verzi, v C++ Builderu 3, se handler ošetřující událost OnCreate volal až po provedení všech inicializací.

Kreslení obrázku fraktálu (soubor okno1.cpp)

Uložení obrázku

Při ukládání obrázku do souboru využijeme komponenty dlg_ulozjako (standardní dialogové okno pro ukládání obrázků). Toto okno bude uživateli nabízet nějaká implicitní jména, např. Obr0.bmp, Obr1.bmp atd. Jako implicitní příponu nastavíme bmp. Nakonec nastavíme příznak, že soubor byl uložen (proměnné ulozeno přiřadíme true).

```
void __fastcall THlavniOkno::mn_ulozClick(TObject *Sender)
{
  static int pocitadlo = 0;       // Pro vytvoření implicitního jména
  AnsiString jmeno = AnsiString("Obr") + AnsiString(pocitadlo++) + ".bmp";
  dlg_ulozjako -> FileName = jmeno;          // Nastav implicitní jméno
  dlg_ulozjako -> DefaultExt = "bmp";    // Nastav implicitní příponu
  if(dlg_ulozjako -> Execute())              // Pokud uživatel stiskl OK...
  {
    jmeno = dlg_ulozjako -> FileName;
    im_obr -> Picture -> SaveToFile(jmeno);
    ulozeno = true;
  }
}
```

Uložení obrázku do souboru (soubor okno1.cpp)

Nový obrázek

Poručí-li si uživatel, že chce nový obrázek, tj. zadá-li v nabídce příkaz Soubor | Nový..., zjistíme nejprve, zda uložil předchozí obrázek, a pokud ne, zeptáme se, zda ho přece jen nechce uložit. Stiskne-li uživatel tlačítko Ano, uložíme soubor; to jsme už naprogramovali, a proto zavoláme funkci THlavniOkno::mn_ulozClick() a přejdeme na další akce. Stiskne-li tlačítko Ne, přejdeme rovnou na další akce. Stiskne-li Storno, vrátíme se, neboť uživatel vlastně nový obrázek vytvářet nechce.

Dále zjistíme rozměry okna a uložíme si je do složek sirka a vyska. Pak nastavíme rozměry bitové mapy a vypočteme konstanty ax a ay (Také to jsme už naprogramovali —zavoláme funkci THlavniOkno::Nastaveni()). Nakonec vymažeme kreslicí plochu a nastavíme příznak, že obrázek byl uložen (prázdný obrázek není třeba ukládat).

```
void __fastcall THlavniOkno::mn_novyClick(TObject *Sender)
{
  if(!ulozeno)                        // Pokud není opbrázek uložen, zjisti,
  {                                                  // zda se má uložit, a případně ho ulož
   switch(DotazChcesUlozit())
   {
    case IDYES:                       // Ano: Zavolej handler, který to má na starosti
      mn_ulozClick(Sender);
      break;
    case IDNO:                        // Ne: Pokračuj
      break;
    case IDCANCEL:                        // Storno: Vrať se, nepokračuj
      return;
   }
  }
  sirka = im_obr->Width;              // Zjisti rozměry okna, nastav rozměry bitové
  vyska = im_obr->Height;       // mapy a vypočti konstanty
  Nastaveni();                              // pro přepočet souřadnic
  im_obr->Canvas->FillRect(ClientRect);     // Vymaž kreslicí plochu
  ulozeno = true;                     // Nastav příznak, že obrázek byl uložen
}
```

Příprava pro kreslení nového obrázku (soubor okno1.cpp)

Při mazání kreslicí plochy jsme použili známou metodu FillRect(). Tato funkce očekává jako parametr hodnotu typu Rect. My jsme použili ClientRect, vlastnost třídy TForm a některých dalších komponent, která obsahuje obdélník představující klientskou oblast okna.

Nastavení

Zbývají nám poslední dvě položky nabídky, O programu... a Nastavení.... Vytvoření dialogového okna O programu přenechám prozatím vaší vynalézavosti — v průběhu vylepšování tohoto programu se k němu ještě vrátíme — a podíváme se na nastavování parametrů programu. Je jasné, že budeme muset vytvořit dialogové okno, ve kterém zadáme souřadnice oblasti, kterou chceme zobrazit, a počet iterací.

Protože plánujeme v budoucnu svůj program rozšířit o další volby a další nastavení, vytvoříme si proto rovnou předpoklady: uděláme dialogové okno s „oušky“.

Začneme tím, že k našemu projektu připojíme nový formulář, který pojmenujeme[33] dlg_nastaveni. Upravíme jeho velikost, do titulku (vlastnost Caption) napíšeme Nastavení a vlastnosti BorderStyle přiřadíme hodnotu bsDialog (nepůjde měnit velikost okna).

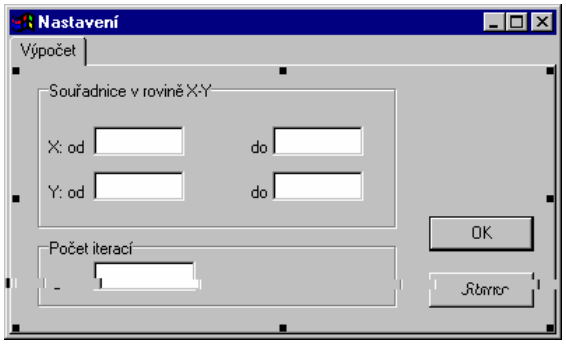
Do vizuálního návrhu tohoto okna vložíme komponentu PageControl, kterou najdeme na paletě Win32; pojmenujeme ji pgc. Tato komponenta ponese jednotlivé „karty s oušky“; zatím ovšem žádnou neobsahuje.

Vedle ní umístíme dvě tlačítka, tj. komponenty Button. (*Nikoli na ni!*). Obě tlačítka vybereme, stisknutím pravého tlačítka myši vyvoláme příruční nabídku a v ní zadáme příkaz Bring To Front. Tím zaručíme, že tlačítka budou *před* kartami, nikoli na některé z nich.[34]

Nyní vybereme komponentu pgc a její vlastnosti Align dáme hodnotu alClient, takže zabere celou klientskou plochu okna. Klikneme na ni pravým tlačítkem myši a v příručním menu, které se poté objeví, zvolíme New Page — nová stránka. Tím vytvoříme novou kartu. Obě tlačítka budou i nadále viditelná.

Každá karta je samostatná komponenta typu TTabSheet a inspektor objektů nám ihned nabídne možnost měnit její vlastnosti. Pojmenujeme ji Strana1 (vlastnost Name) a do „ouška“ napíšeme Výpočet (vlastnost Caption).

Na tuto kartu pak vložíme dvě komponenty GroupBox, které najdeme na paletě Standard. Poslouží nám jako ohraničení dalších řídicích prvků, v našem případě editačních polí (obr. 6.3). Jejich velikost upravíme tak, aby se do nich potřebné prvky vešly.



Obr. 6.3 Vizuální návrh dialogového okna pro nastavení parametrů programu

U prvního „skupinového boxu“ uložíme do vlastnosti Caption řetězec Souřadnice v rovině X-Y, u druhé řetězec Počet iterací. Tím určíme nápisy v záhlaví skupin.

Do první skupiny pak vložíme čtyři editační pole, tj. komponenty Edit, a pojmenujeme je ed_xlh, ed_ylh, ed_xpd a ed_ypd. Budou sloužit k zadávání rozsahu souřadnic v komplexní rovině: V poli ed_xlh uvedeme souřadnice *x* levého horního rohu zkoumané oblasti, v poli ed_ypd souřadnice pravého dolního rohu atd. K nim doplníme vysvětlující nápisy (komponenty Label) tak, jako na obr. 6.2.

---

[33] Prostřednictvím vlastnosti Name v inspektoru objektů. Připomeňme si, že „pojmenovat“ komponenty nebo formulář vlastně znamená pojmenovat ukazatel na ni, neboť instance tříd z knihovny VCL musí být alokovány dynamicky.

[34] Zatím se může zdát, že postupujeme zbytečně složitě — stačilo by obě tlačítka vložit normálně na kartu. Tento postup oceníme později, až do dialogového okna přidáme další karty a budeme chtít, aby tlačítka OK a Storno byla dostupná na všech. Poznamenejme, že karty (tedy komponenty TTabSheet se chovají jako kontejnery na komponenty. To znamená, že komponenta, kterou položíme na kartu, je touto kartou vlastněna a my ji např. už nemůžeme přemístit mimo ni, přenést před další kartu apod. Jestliže ovšem tlačítko položíme nejprve mimo kartu a pak ho nad ni přeneseme, nebude ho žádná karta vlastnit a bude nad ní.

Do druhé skupiny, nadepsané Počet iterací, vložíme jedinou komponentu Edit, kterou pojmenujeme ed_pocit; budeme v ní určovat počet iterací.

Nakonec nastavíme vlastnosti obou tlačítek. První z nich pojmenujeme tl_ok (vlastnost Name), napíšeme na ně OK (vlastnost Caption), vlastnost Default nastavíme na true, takže bude reagovat i na stisknutí klávesy ENTER, a vlastnosti ModalResult přiřadíme mrOk, takže stisknutí tohoto tlačítka způsobí uzavření dialogového okna a funkce ShowModal() v tom případě vrátí hodnotu IDOK.

Druhé tlačítko pojmenujeme tl_cancel, napíšeme na ně Storno, vlastnost Cancel nastavíme na true, takže bude reagovat i na stisknutí klávesy ESC, a vlastnosti ModalResult přiřadíme mrCancel, takže stisknutí tohoto tlačítka způsobí uzavření dialogového okna a funkce ShowModal() přitom vrátí hodnotu IDCANCEL.

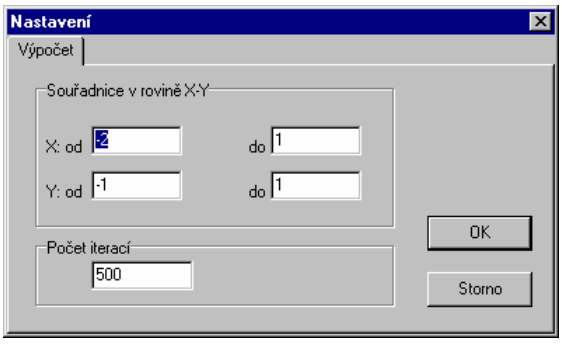
Inicializace

Při otevření tohoto dialogového okna by se v něm měla automaticky zobrazit platná nastavení. O to se postaráme při události OnShow, tedy při zobrazení okna. Připomeňme si, že text zobrazený v editačním poli je hodnotou vlastnosti Text. Potřebná data zjistíme pomocí přístupových funkcí definovaných ve třídě hlavního okna.

```
void __fastcall TDlg_nastaveni::FormShow(TObject *Sender)
{
  ed_xpd->Text = HlavniOkno->xpd();
  ed_xlh->Text = HlavniOkno->xlh();
  ed_ylh->Text = HlavniOkno->ylh();
  ed_ypd->Text = HlavniOkno->ypd();
  ed_pocit->Text = HlavniOkno->PocetIteraci();
  FocusControl(ed_xlh);
}
```

Při zobrazení dialogového okna Nastavení se v něm objeví platné hodnoty (soubor nastav.cpp)

Po zadání příkazu Obrázek | Nastavení tedy uvidíme např. dialogové okno z obr. 6.4.



Obr. 6.4 Po vyvolání ukazuje dialogové okno Nastavení právě platné hodnoty

Poznámka

Funkce HlavniOkno->xpd() a další vracejí čísla typu double. Vlastnost TEdit::Text je typu AnsiString, a protože tato třída má mj. konstruktor s parametrem typu double, konvertuje se přiřazovaná hodnota automaticky. Příkaz

```
ed_xpd->Text = HlavniOkno->xpd();
```

tedy znamená totéž jako

```
ed_xpd->Text = AnsiString(HlavniOkno->xpd());
```

Získání dat z dialogového okna

Po stisknutí tlačítka OK se dialogové okno uzavře, hodnoty v něm nastavené se ale neztratí. V handleru, který toto okno vyvolal, je přečteme. Nejprve zjistíme, zda uživatel stiskl tlačítko OK. Pokud ano, tj. pokud funkce ShowModal() vrátila IDOK, uložíme tyto hodnoty do odpovídajících datových složek a vypočteme nové hodnoty konstant ax a ay. K převodu řetězce typu AnsiString na reálná, resp. celá čísla použijeme metody ToDouble() a ToInt().

```
void __fastcall THlavniOkno::mn_nastaveniClick(TObject *Sender)
{
  if(dlg_nastaveni->ShowModal()==IDOK)
  {
   XLH = dlg_nastaveni -> ed_xlh -> Text.ToDouble();
   XPD = dlg_nastaveni -> ed_xpd -> Text.ToDouble();
   YLH = dlg_nastaveni -> ed_ylh -> Text.ToDouble();
   YPD = dlg_nastaveni -> ed_ypd -> Text.ToDouble();
   VypoctiKonstanty();
   pocetIteraci = dlg_nastaveni -> ed_pocit -> Text.ToInt();
  }
}
```

Po uzavření dialogového okna Nastavení (soubor okno1.cpp)

Smíme skončit?

Může se stát, že uživatel dá příkaz k ukončení programu a zapomene uložit vytvořený obrázek. Protože jde o výsledek časově náročného výpočtu, je rozumné, aby se program v takovém případě zeptal, zda si uživatel opravdu přeje skončit. O to se postará handler ošetřující událost OnCloseQuery.

```
void __fastcall THlavniOkno::FormCloseQuery(TObject *Sender, bool &CanClose)
{
  if(!ulozeno)                          // Je-li obrázek uložen, nedělej nic,
  {
   switch(DotazChcesUlozit())      // jinak se zeptej, zda se má obrázek uložit.
   {
    case IDYES:                         // Ano: Zavolej funkci pro uložení, a pokud
     mn_ulozClick(Sender);          // uživatel obrázek uložil, skonči, jinak ne
     CanClose = ulozeno;
     break;
    case IDNO:                              // Ne: klidně skonči
     CanClose = true;
     break;
    case IDCANCEL:                      // Storno: nesmíš skončit
     CanClose = false;
     break;
   }
  }
  else CanClose = true;
}
```

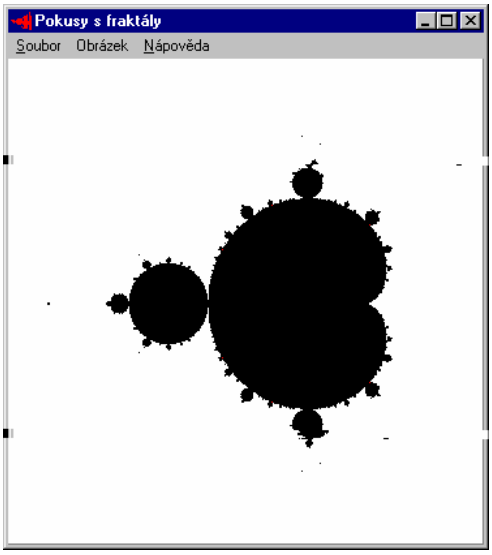
Handler, který se připomíná uložení obrázku před skončením programu (soubor okno1.cpp)

Jestliže obrázek není uložen a na dotaz, zda ho chce uložit, odpoví uživatel Ano, zavolá se funkce mn_ulozClick(). Ta vyvolá dialogové okno pro uložení souboru, a pokud v něm uživatel stiskne tlačítko Uložit, obrázek se uloží a program skončí. Stiskne-li v okně pro uložení souboru tlačítko Storno, obrázek se neuloží a program neskončí.

Jestliže na dotaz, zda se má obrázek uložit, odpoví uživatel stiskem tlačítka Storno, program neskončí. Odpoví-li Ne, obrázek se neuloží a program skončí.

Výsledek

Program v této podobě najdete na doprovodném CD v adresáři KAP06\01. Až jej spustíte, uvidíte po chvíli něco podobného jako na obrázku 6.5. Nebuďte ale překvapeni, že se chvíli nebude nic dít a bude se zdát, že program „zamrzl“ — nebude reagovat na příkazy z nabídky, nepůjde přemístit jeho okno, nepůjde jej ukončit atd. Po chvilce se v okně objeví obrázek a vše bude v pořádku. V následujícím oddílu si ukážeme, jak tyto a další vady odstranit.



Obr. 6.5 Mandelbrotova množina (osamělé černé tečky nejsou chyby tiskárny, patří k množině)

6.2 Verze 2.0: Odstraňujeme nejhrubší chyby

Náš program sice funguje, ale má mnoho nedostatků. O nejnápadnějším z nich jsme již hovořili v závěru předchozího oddílu — po dobu výpočtu nekomunikuje s okolím. Programátoři mají pro takové programy řadu pojmenování, převážně neslušných, a uživatelé se na ně dívají jako na důkaz programátorské neschopnosti.

Další problém se týká chyb při zadávání souřadnic v dialogovém okně Nastavení. Jestliže uživatel do některého z editačních polí napíše znakový řetězec, který nelze interpretovat jako číslo, objeví se chybové hlášení, které ukazuje obr. 6.6. Při spuštění z prostředí C++ Builderu uvidíme okno, které nám oznámí, že v procesu vznikla výjimka, a nabídne nám ladění na úrovni instrukcí strojního kódu. Při běhu mimo prostředí po tomto chybovém hlášení zmizí dialogové okno Nastavení.

Obr. 6.6 Stačí např. zapomenout, že v českých Windows se používá desetinná čárka, nikoli tečka

Ještě horší bude, jestliže uživatel uvede záporný počet iterací. Program to klidně přijme, nepozná, že jde o chybu, ale nic nespočítá — funkci Vypocet() se budou všechny body jevit jako prvky Mandelbrotovy množiny, neboť cyklus, který počítá prvky posloupnosti (1), nikdy neproběhne. Je tedy třeba zařídit kontrolu vstupu, a to pokud možno hned při zadávání.

To ale není všechno. Výpočet fraktálu je poměrně zdlouhavý, zejména pokud máme pomalejší počítač a zadáme rozsáhlejší obrázek, tj. větší okno. To znamená, že uživatel by měl mít možnost výpočet kdykoliv přerušit, aniž by musel násilně ukončit běh programu.

Komunikace s běžícím programem

Začneme nejvážnějším problémem — tím, že program s námi po dobu výpočtu nekomunikuje a že ani nezobrazuje průběžný stav, i když každý vypočtený bod ihned nakreslíme (nebo si to alespoň myslíme). V čem je problém?

Operační systém Windows komunikuje s programy pomocí zpráv. Jestliže myší klikneme na příkaz menu, Windows to zjistí a pošlou o tom našemu programu zprávu. Tuto zprávu uloží do fronty, ze které si ji náš program vyzvedne — ovšem až na to bude mít čas. C++ Builder na tom nic nezměnil, jen tuto skutečnost před námi zakryl — až dosud jsme se s ní setkali pouze ve 3. kapitole při práci s časovačem.

Vraťme se k našemu programu. Ve chvíli, kdy příkazem Obrázek | Výpočet spustíme výpočet jednotlivých bodů, začne program počítat a nemá možnost starat se o zpracovávání zpráv. Proto nereaguje na příkazy z nabídky, na klávesové zkratky nebo na pokusy o přemístění okna. Proto se také nezobrazuje průběžný stav obrázku. Celý děj probíhá přibližně takto: Po vykreslení nového bodu, tj. po přiřazení

```
im_obr -> Canvas -> Pixels[ix][iy] = Vypocet(X(ix), Y(iy), pocetIteraci);
```

se komponenta Image postará v rámci svých možností o překreslení výstupu, tj. sdělí operačnímu systému, že obsah okna je neplatný. Operační systém na to zareaguje tím, že pošle programu zprávu WM_PAINT, což je žádost o překreslení obsahu okna. Ovšem tato zpráva zůstane ve frontě, neboť náš program počítá a počítá…

Existuje několik způsobů řešení.

- ✧ Můžeme použít časovač. Nastavíme ho na krátký interval, např. 1 ms, a při každé události OnTimer spočítáme nějakou malou část, např. jeden pixel nebo jeden sloupec pixelů.
- ✧ Můžeme použít funkci TApplication::ProcessMessages(), která dává možnost přerušit výpočet a zpracovat zprávy, které se ve frontě nahromadily.
- ✧ Můžeme výpočet přesunout do jiného vlákna (threadu) a nechat ho běžet paralelně s vláknem, které se stará o uživatelské rozhraní programu.

První možnost, použití časovače, je asi nejméně vhodná. Představte si nevelký obrázek, např. 300 × 200 pixelů. Pro něj musíme vypočítat barvu v „pouhých“ 60000 bodech. I kdybychom nastavili interval 1 ms a události OnTimer skutečně po 1 milisekundě nastávaly, znamenalo by to prodloužení doby výpočtu o 60 vteřin, tedy 1 minutu. Ve skutečnosti ale bude zdržení mnohem větší.

V tomto oddílu se podíváme na druhou možnost, použití funkce TApplication::ProcessMessages(), a ve verzi 3.0 (oddíl 6.4) se podíváme i na třetí možnost, multihtreadovou aplikaci.

Metoda ProcessMessages()

Metoda ProcessMessages() je definována ve třídě TApplication. Jestliže ji zavoláme, pozastaví se výpočet a aplikace dostane příležitost zpracovat zprávy, které čekají ve frontě. Jde tedy o to, kdy máme metodu ProcessMessages() volat. Pokud bychom ji volali vždy po zjištění barvy každého pixelu, tj. pokud bychom cyklus v metodě THlavniOkno::mn_vypocetClick() upravili do tvaru

```
for(int ix = 0; ix < sirka; ix++)
{
  for(int iy = 0; iy < vyska; iy++)
  {
    im_obr -> Canvas -> Pixels[ix][iy] = Vypocet(X(ix), Y(iy), pocetIteraci);
    Application -> ProcessMessages();
  }
}
```

byl by výpočet zbytečně pomalý, neboť kvůli každému bodu by se vlastně překresloval celý obrázek. Zařadíme tedy její volání do vnějšího cyklu, tj. po vypočtení jedné svislé řady pixelů:

```
for(int ix = 0; ix < sirka; ix++)
{
  for(int iy = 0; iy < vyska; iy++)
  {
    im_obr -> Canvas -> Pixels[ix][iy] = Vypocet(X(ix), Y(iy), pocetIteraci);
  }
  Application -> ProcessMessages(); // Zpracujeme nahromaděné zprávy
}
```

Přerušení výpočtu

Nyní už také můžeme vyřešit požadavek na přerušení výpočtu. Použijeme asi nejjednodušší možnost: Vytvoříme si pomocnou proměnnou a příkazem z nabídky do ní uložíme hodnotu, která bude znamenat příkaz k ukončení výpočtu. Handler mn_vypocetClick() bude hodnotu této proměnné v pravidelných intervalech kontrolovat.

Nejprve tedy přidáme do třídy THlavniOkno datovou složku prerus_vypocet typu bool a v konstruktoru jí přiřadíme počáteční hodnotu false. Pak přidáme do nabídky Obrázek novou položku Přeruš; odpovídající komponentu typu TMenuItem uložíme pod jménem mn_prerus.

Handler ošetřující vybrání této položky bude velice jednoduchý — uloží do prerus_vypocet hodnotu true a tím nastaví příznak, že výpočet má předčasně skončit:

```
void __fastcall THlavniOkno::mn_prerusClick(TObject *Sender)
{
  prerus_vypocet = true;
}
```

Poté znovu upravíme handler mn_vypocetClick(), který se stará o výpočet bodů obrázku — do vnějšího cyklu přidáme kontrolu proměnné prerus_vypocet. Jestliže tato proměnná obsahuje true, změníme její hodnotu na false a ukončíme cyklus:

```
for(int ix = 0; ix < sirka; ix++)
{
 for(int iy = 0; iy < vyska; iy++)
 {
  im_obr -> Canvas -> Pixels[ix][iy] = Vypocet(X(ix), Y(iy), pocetIteraci);
 }
 Application -> ProcessMessages();
 if(prerus_vypocet)                // Je třeba skončit?
 {                                                  // pokud ano,
  prerus_vypocet = false; // zruš příznak
  break;                                        // a opusť cyklus
 }
}
```

Úprava nabídky

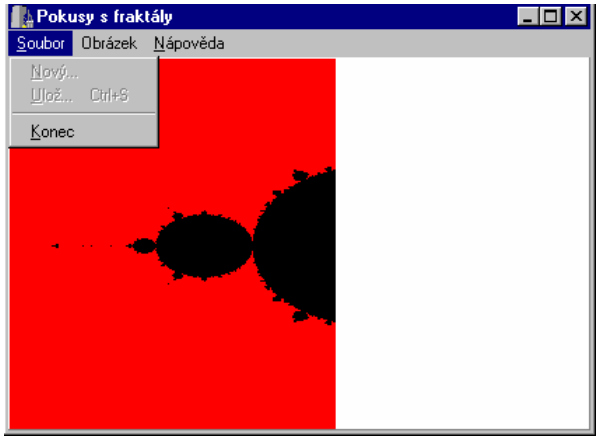
Tím jsme odstranili dvě z vad, o kterých jsme hovořili v úvodu — program s námi komunikuje s dostatečně rychlou odezvou a lze jej uprostřed výpočtu přerušit. Ovšem okamžitě se objeví další problémy: Co se stane, jestliže před zahájením výpočtu (omylem nebo schválně) použijeme příkaz Obrázek | Přeruš? Co se stane, jestliže uprostřed výpočtu použijeme příkaz Soubor | Nový? Co když se pokusíme ještě nedokreslený obrázek uložit? (Vyzkoušejte si to.)

Bylo by tedy vhodné některé příkazy v průběhu výpočtu zakázat a po skončení je zase povolit a naopak, příkaz Přeruš povolit jen v průběhu výpočtu. Toho dosáhneme přiřazením hodnoty true, resp. false vlastnosti Enabled odpovídající komponentě typu TMenuItem. Nejprve tedy vyhledáme v inspektoru objektů komponentu mn_prerus a nastavíme její vlastnost Enabled na false. Pak přidáme na začátek a na konec handleru mn_vypocetClick() příkazy, které se postarají o úpravy nabídek.

```
void __fastcall THlavniOkno::mn_vypocetClick(TObject *Sender)
{
  ulozeno = false;                  // Obrázek ještě nebyl uložen
  mn_prerus ->Enabled = true;       // Uprav použitelnost položek menu
  mn_vypocet->Enabled = false;
  mn_novy->Enabled = false;
  mn_uloz->Enabled = false;
  for(int ix = 0; ix < sirka; ix++)          // Vlastní výpočet
  {
   for(int iy = 0; iy < vyska; iy++)
   {
    im_obr -> Canvas -> Pixels[ix][iy] = Vypocet(X(ix), Y(iy), pocetIteraci);
   }
    Application -> ProcessMessages();       // Umožni zpracovat zprávy
    if(prerus_vypocet)                      // Je třeba přerušit výpočet?
   {                                                        // Pokud ano,
    prerus_vypocet = false;                 // zruš příznak
    break;                                          // a opusť cyklus
   }
  }
  mn_prerus ->Enabled = false;              // V každém případě zruš
  mn_vypocet->Enabled = true;                       // úpravy použitelnosti menu provedené
  mn_novy->Enabled = true;                  // na počátku
  mn_uloz->Enabled = true;
}
```

Upravený handler, který má na starosti výpočet bodů obrázku (soubor okno1.cpp)

Po této úpravě již nebude možné nesmyslné příkazy zadat, viz obr. 6.7.



Obr. 6.7 Příkazy, které by mohly působit problémy, jsme zakázali

Chybný vstup

Chybové hlášení z obr. 6.6 mají na svědomí metody pro převod řetězců na čísla AnsiString::ToDouble() a AnsiString::ToInt(). Jestliže tyto metody narazí v řetězci na znak, který nepatří do reprezentace reálného, resp. celého čísla, vyvolají výjimku typu EConvertError. Mohli bychom se tedy pokusit zachytit tuto výjimku a vyzvat uživatele k nápravě.

Ve skutečnosti je ale v okamžiku uzavření dialogového okna již pozdě. Vhodnější je upozornit uživatele, že se spletl, ihned po zadání textu do editačního pole. K tomu můžeme využít buď událost OnChange, která nastane při změně textu v editačním poli, nebo události OnExit, která nastane, když editační pole ztratí fokus. Vhodnější je druhá možnost, neboť nastane až poté, co uživatel dokončí zadávání dat do editačního pole.

Mohli bychom tedy vytvořit následující handler:

```
void __fastcall Tdlg_nastaveni::ed_xlhExit(TObject *Sender)
{
  try
  {
   ed_xlh->Text.ToDouble();
  }
  catch(...)
  {
   // Upozorni uživatele na chybu
  }
}
```

Zde se prostě pokusíme o konverzi. Pokud se podaří, je číslo zadáno správně, program přeskočí blok catch a handler bez problémů skončí. Pokud ne, nastane výjimka, řízení přejde do bloku catch a handler upozorní uživatele na chybu.

Ovšem téměř stejný text budeme psát i pro zbývající tři editační pole ve skupině Souřadnice v rovině X-Y (bude se lišit jen jménem editačního pole). Bude tedy rozumné jeden napsat handler tak, abychom jej mohli použít pro všechna čtyři editační pole. Parametr Sender sice obsahuje adresu komponenty, která událost způsobila, je ovšem typu TObject*. Musíme jej tedy přetypovat.

```
void __fastcall Tdlg_nastaveni::ed_xlhExit(TObject *Sender)
{
  TEdit * ed = dynamic_cast<TEdit*>(Sender);      // Přetypujeme
  if(ed)                                                    // a pokud se přetypování podaří...
  {
    try
    {
      ed->Text.ToDouble();                          // Zkus převod na číslo
    }
    catch(EConvertError&)                           // a když se to nepovede,
    {
      Application -> MessageBox(                    // vypiš upozornění
              (ed->Text+" není správně zapsané číslo." Dosazuji 0.").c_str(),
                 "Chyba", MB_OK);
      ed->Text = 0;                                 // dosaď implicitní hodnotu
      FocusControl(ed);                       // a vrať tam fokus
    }
  }
```

```
}
```

Kontrola správnosti čísla zapsaného v editačním poli

Tento handler použijeme pro událost OnExit u komponent ed_xlh, ed_xpd, ed_ylh a ed_Ypd. Handler pro editační pole ed_pocit bude velmi podobný, bude však navíc kontrolovat, zda je zadaný počet iterací alespoň 10. Pokud není, vyvolá výjimku typu ERangeError:

```
void __fastcall TdIg_nastaveni::ed_pocitExit(TObject *Sender)
{
  TEdit * ed = dynamic_cast<TEdit*>(Sender);
  if(ed)
  {
    try
    {
            int i = ed->Text.ToInt();                   // Zkus převod na číslo, a když se
            if(i < 10) throw ERangeError("");   // povede, zjisti, zda je alespoň 10
    }
    catch(EConvertError&)                       // Když se nepovede převod,
    {                                                   // postupuj jako minule
            Application -> MessageBox(
                    (ed->Text+" není správně zapsané číslo. Dosazuji 10.").c_str(),
                    "Chyba", MB_OK);
            ed->Text = 10;
            FocusControl(ed);
    }
    catch(ERangeError&)                         // Když je číslo < 10,
    {                                                   // vypiš upozornění,
            Application -> MessageBox(      "Počet iterací menší než 10. Dosazuji 10.",
                                            "Chyba", MB_OK);
            ed->Text = 10;                      // dosaď implicitní hodnotu
            FocusControl(ed);                   // a vrať fokus
    }
  }
}
```

Poznámky

- ✧ Operátor dynamic_cast slouží k přetypování mezi objektovými typy, případně mezi ukazateli na ně. Přitom kontroluje, zda je požadované přetypování možné, tj. zda jde o přetypování z předka na potomka nebo naopak. Pokud požadované přetypování není možné, vrátí 0. Proto jsme všechny následující operace uzavřeli do bloku if. Pokud by se ovšem přetypování nepodařilo, znamenalo by to, že jsme tento handler někde použili nesprávně.
- ✧ V bloku catch vypíšeme upozornění, dosadíme do editačního pole implicitní hodnotu a přeneseme na ně fokus voláním funkce FocusControl().
- ✧ Ani toto řešení není dokonalé — bude fungovat, jestliže po zadání nesprávné hodnoty přejdeme do jiného editačního pole pomocí myši nebo tabulátoru nebo stiskneme-li myší tlačítko OK. Nebude ale fungovat v případě, že po zadání nesprávné hodnoty stiskneme klávesu ENTER. V tom případě totiž nenastane událost OnExit. Chceme-li odstranit i tuto vadu, musíme vyvolat událost OnExit např. tím, že při uzavření okna (při události OnClose) přenesete fokus na některou jinou komponentu, např. na kartu:

```
void __fastcall TdIg_nastaveni::FormClose(TObject *Sender, TCloseAction &Action)
{
  FocusControl(Strana1);
}
```

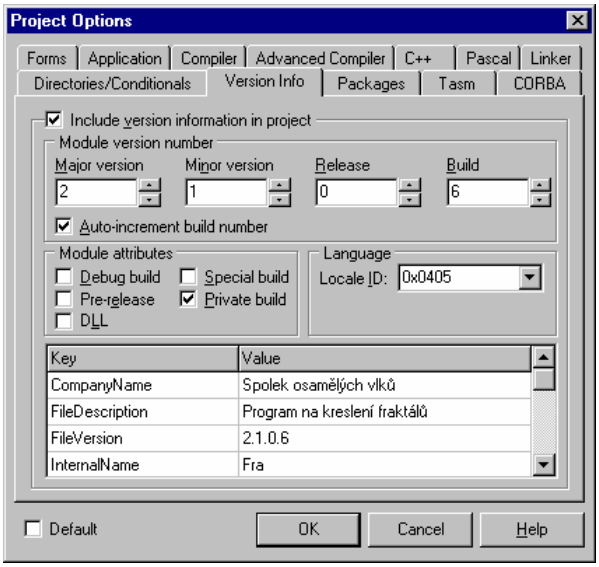
Na doprovodném CD najdete tento program v adresáři KAP06\02 v poněkud rafinovanější úpravě, která zajistí, že při zadání špatné hodnoty a stisknutí ENTER se dialogové okno po vypsání chybové zprávy neuzavře. Toto řešení využívá události OnCloseQuery a pomocné proměnné lze_zavrit.

6.3 Verze 2.1: Drobná vylepšení

Za verzi 2.0 se už nemusíme stydět — na trhu se bohužel setkáme i s podstatně horšími programy. Přesto se pokusíme o další vylepšení, drobné úpravy, které našemu programu dodají trochu lesku.

Automatické číslování verzí v okně O programu

Jako první se naučíme využívat automatické číslování verzí. Při prohlídce prostředí jste si možná všimli, že v dialogovém okně Project | Options je mj. karta Version Info (obr. 6.8), která umožňuje uložit do aplikace informace o číslu verze, jménu společnosti, držiteli autorského práva atd.



Obr. 6.8 Volby pro automatické ukládání čísla verze a dalších informací do aplikace

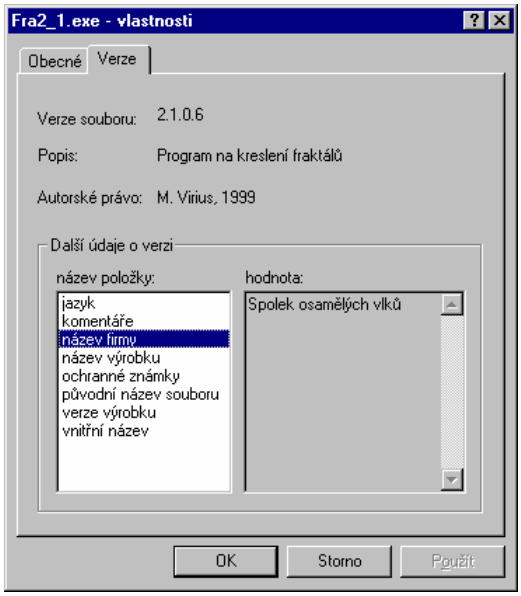
Jestliže ve Windows kliknete pravým tlačítkem myši na ikonu programu a z příruční nabídky vyberete Vlastnosti, objeví se tyto informace na kartě Verze (obr. 6.9). (Musí jít o ikonu souboru, nikoli o zástupce.)

Tyto informace se uloží do spustitelného souboru jako prostředky (resource). Vzhledem k tomu, že podobné informace bývají pravidelně součástí dialogového okna O programu, bylo by rozumné umět je za běhu programu získat a použít. Jde to a my si ukážeme jak.

Poslouží nám funkce GetFileVersionInfoSize(), GetFileVersionInfo() a VerQueryValue() z Windows API. První z nich vrátí velikost pole potřebného pro uložení informací o verzi, druhá uloží do zadaného pole informace o verzi a třetí z nich vyjme potřebnou část (např. číslo verze, jméno firmy apod.).

Protože dialogové okno O programu jsme již několikrát spolu vytvořili, budu předpokládat, že ho máte již hotové a chcete do něj přidat informaci o čísle verze; toto dialogové okno se jmenuje Oprogramu a zdrojový kód, který se o něj stará, je uložen v souboru oprog.cpp.

Do tohoto okna — tedy do jeho vizuálního návrhu — umístíme komponentu Label, která bude mít za úkol zobrazovat číslo verze, a nastavíme vhodné vlastnosti (velikost a barvu písma atd.). Číslo verze zjistíme při vytváření tohoto okna, tj. v handleru události OnCreate.



Obr. 6.9 Informace o verzi souboru ve Windows

Postup ukazuje následující výpis.

```
void __fastcall TOprogramu::FormCreate(TObject *Sender)
{
  Label2->Caption = "";             // Implicitně prázdný nápis
  unsigned long m;                  // Pomocná proměnná bez významu, ale nutná
  int n = GetFileVersionInfoSize(Application->ExeName.c_str(), &m);
  if(n)                                          // Pokud je VersionInfo k dispozici...
  {
   char *Info = new char [n+1];   // Alokujeme pole pro uložení VersionInfo
   char *cisver;                  // Ukazatel na pole obsahující číslo verze
   unsigned vel=0;                // Velikost pole s číslem verze
   int ii= GetFileVersionInfo(Application->ExeName.c_str(), 0, n, Info);
   if(ii)                                    // Pokud se operace nepodařila, konec
   {
    ii = VerQueryValue(Info, // Zjisti číslo verze
                       TEXT("\\StringFileInfo\\040504E2\\FileVersion"),
                       (void**)&cisver, &vel);
              if(ii)                                  // Pokud se to podařilo
              {                                                // vytvoř řetězec "Verze x.x.x.x"
                  AnsiString pom = AnsiString("Verze ") + cisver;
                  Label2->Caption = pom;    // a zobraz ho v komponentě. Zjisti velikost
                  Canvas -> Font = Label2 -> Font;          // nápisu a vycentruj ho
                  Label2->Left = (ClientWidth - Canvas->TextWidth(pom))/2;
              }
   }
   delete Info;                      // Nakonec uvolni paměť
  }
}
```

Zjištění čísla verze při vytvoření dialogového okna O programu (soubor oprog.cpp)

Ve skutečnosti nejde o nic složitého:

- ✧ Nejprve pomocí funkce GetFileVersionInfoSize() zjistíme velikost paměti potřebné pro uložení informací o verzi a uložíme si ji do proměnné n. Prvním parametrem této funkce je jméno souboru, o němž zjišťujeme informace; to zjistíme pomocí vlastnosti Application->ExeName. Pomocná proměnná m nemá význam, musíme ji ale uvádět.
- ✧ Pokud nejsou informace o verzi k disposici, vrátí tato funkce 0, proto jsme následující operace uzavřeli do příkazu if.
- ✧ Pak alokujeme pole t, do kterého se informace o verzi překopírují. Proměnná cisver bude obsahovat ukazatel na výsledek, tedy v našem případě na řetězec obsahující číslo verze. Proměnná vel bude obsahovat velikost výsledného řetězce.
- ✧ Pak zavoláme funkci GetFileVersionInfo(). Jejím prvním parametrem je opět jméno souboru, druhý nemá význam, takže můžeme použít 0. Třetím parametrem je velikost pole, do kterého se mají

informace o verzi uložit, a čtvrtým je ukazatel na toto pole. Pokud funkce GetFileVersionInfo() uspěje, vrátí nenulovou hodnotu a do pole t uloží výsledek, jinak vrátí 0.

- ✧ Pokud tato funkce neuspěje, vrátí 0, a proto skončíme: Vše ostatní jsme uzavřeli do příkazu if.
- ✧ Pak zavoláme funkci . Jejím prvním parametrem je pole Info, do kterého nám uložila potřebné informace funkce GetFileVersionInfo(). Druhým parametrem je řetězec, který specifikuje, jakou informaci chceme, třetím parametrem je adresa ukazatele na první prvek pole, do kterého se uloží výsledek, a posledním parametrem je adresa proměnné, do které se uloží velikost výsledku. Pokud tato funkce uspěje, vrátí nenulovou hodnotu, jinak vrátí 0.
- ✧ Pokud se podařilo získat potřebné informace, vytvoříme žádaný nápis, uložíme ho do vlastnosti Caption komponenty Label, zjistíme velikost tohoto nápisu a komponentu umístíme ve vodorovném směru do středu okna.
- ✧ Druhý parametr funkce je znakový řetězec složený ze tří částí oddělených obráceným lomítkem. První říká, že chceme informace uložené v podobě znakového řetězce, druhá určuje jazyk a kódovou stránku a třetí říká, kterou položku z informací o verzi chceme. Tento řetězec musíme pomocí makra TEXT transformovat do kódu UNICODE, neboť tak jsou informace o verzi uloženy.
- ✧ Údaj o jazyku se skládá ze dvou částí. První čtyři znaky tvoří hexadecimální kód jazyka (pro češtinu je to 0405, pro slovenštinu 04B1, pro angličtinu 0409). Druhé čtyři znaky tvoří hexadecimální vyjádření čísla kódové stránky (pro nás 04E2, pro angličtinu 04E4). Číslo aktuálně nastaveného jazyka najdete v okně Program | Options na kartě Version Info v poli Locale ID (obr. 6.8). Pokud používáte Windows NT, najdete čísla všech lokálních nastavení v souboru intl.inf v podadresáři INF domovského adresáře Windows.
- ✧ Jako poslední část můžeme uvést vedle FileVersion také CompanyName, FileDescription, InternalName, ProductName, ProductVersion a další — podorbnosti najdete v nápovědě k této funkci.
- ✧ Řetězec obsahující číslo verze se skládá ze 4 položek oddělených tečkami: Jde o hlavní číslo, vedlejší číslo, třetí číslo a číslo sestavení. Pokud bychom chtěli použít jen některé části, musíme si tento řetězec rozložit sami, např. pomocí standardní funkce strtok().

Stavový řádek

Lepší programy mají zpravidla také tzv. stavový řádek. To je pole u spodního okraje okna, ve kterém se zobrazují stavové informace (např. postup výpočtu, nápověda apod.). K tomuto účelu nabízí C++ Builder komponentu StatusBar, kterou najdeme na paletě Win32.

Vybereme tedy tuto komponentu a vložíme ji do vizuálního návrhu hlavního okna. Automaticky se umístí ke spodnímu okraji, neboť její vlastnost Align má implicitní hodnotu alBottom.

V inspektoru objektů vyhledáme vlastnost SimplePanel a nastavíme ji na true. Tím způsobíme, že se tato komponenta bude chovat jako jednoduchý panel — bude zobrazovat pouze text, který je obsahem vlastnosti SimpleText.[35]

Nyní už zbývá jediné — zařídit, aby se ve stavovém řádku zobrazovaly správné informace. Půjde nám především o indikaci postupu výpočtu, takže úpravy se budou týkat funkce mn_vypocetClick(). Před volání funkce ProcessMessages() zařadíme příkaz

```
sb_stav->SimpleText = AnsiString("Vypo teno ") + 100*ix/sirka + "%";
```

Také při přerušení nebo dokončení výpočtu bychom měli vypsat odpovídající informace. To znamená, že mezi příkazy, které se provádějí při přerušení, zařadíme také

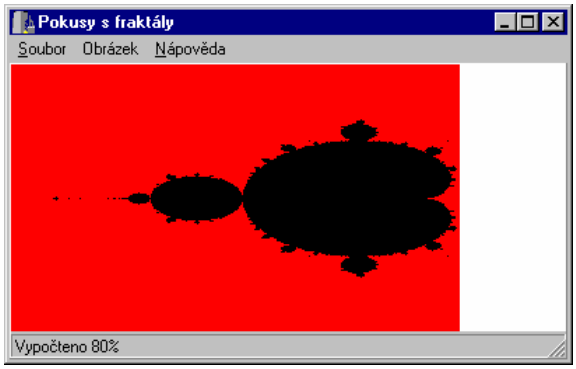
```
sb_stav->SimpleText = "P eru eno";
```

a jako poslední v těle funkce mn_vypocetClick() zařadíme

```
if(sb_stav->SimpleText != "P eru eno") sb_stav->SimpleText="Hotovo";
```

Výsledek ukazuje obrázek 6.10.

[35] Komponenta StatusBar se může chovat i jako stavový řádek složený z několika panelů, se kterými můžeme zacházet prostřednictvím vlastnosti Panels.



Obr. 6.10 Stavová řádka v akci

Rozměry a barvy bitové mapy

V dalším kroku přidáme možnost volit rozměry vytvářené bitové mapy a barevné kombinace, které se při zobrazování fraktálu používají. Budeme je zadávat v dialogovém okně Nastavení na další kartě, které dáme nadpis Bitová mapa.

Rozměry

Nejprve tedy přejdeme do vizuálního návrhu okna Nastavení. Klikneme pravým tlačítkem myši na komponentu PageControl mimo její první stránku a v příruční nabídce vybereme možnost New Page; C++ Builder automaticky přidá novou prázdnou kartu, tedy novou komponentu TabSheet. Pojmenujeme ji Strana2 (vlastnost Name) a dáme jí nadpis Bitová mapa (vlastnost Caption).

Všimněte si, že komponenty, které jsme umístili *na* kartu Strana1, jsou viditelné, pouze je-li tato strana aktivní, zatímco tlačítka, která jsme umístili *nad* komponentu pgc, a tedy i nad všechny karty, jsou viditelná stále.

Na kartu Strana2 položíme komponentu Groupbox, upravíme ji do vhodné velikosti, nadepíšeme ji Rozměry bitové mapy v pixelech (vlastnost Caption) a umístíme do ní dvě komponenty Edit, které pojmenujeme ed_vyska a ed_sirka. Budou sloužit k zadávání požadovaných rozměrů bitové mapy. (Viz obr. 6.11.)

Obsluha těchto dvou editačních polí je podobná jako obsluha pole ed_pocit pro zadávání počtu iterací, a proto se zde nebudeme zabývat podrobnostmi, které jistě zvládnete sami. (Můžete si také vyhledat hotové řešení na doprovodném CD v adresáři KAP06\021.)

Zmíníme se pouze o dvou problémech:

Za prvé, v handleru mn_novyClick() v hlavním okně musíme odstranit příkazy, které nastavují rozměry bitové mapy podle rozměrů okna.

Za druhé, příkaz

```
FocusControl(Strana1);
```

v handleru Tdlg_nastaveni::FormCloseQuery() (pokud jste vyšli z verze 2.0, tak v handleru Tdlg_nastaveni::FormClose()) musíme nahradit příkazem

```
FocusControl(pgc->ActivePage);
```

C++ Builder totiž nedovoluje přenést fokus na neaktivní stránku. (Připomeňme si, že tento příkaz sloužil k vyvolání události OnExit v posledním aktivním editačním poli při uzavírání dialogového okna, takže je jedno, na kterou stránku fokus přeneseme.)

Barvy

Vrátíme se opět k vizuálnímu návrhu dialogového okna Nastavení a přidáme do něho komponentu RadioGroup (skupina přepínačů), kterou najdeme na paletě Standard. Nadepíšeme ji Barevné kombinace (vlastnost Caption). Do ní umístíme dvě komponenty RadioButton, které najdeme opět na paletě Standard. Pojmenujeme je rb_cc a rb_mb a napíšeme k nim Červená — černá a Modrá — bílá. To budou barevné kombinace, které budou označovat.

Poznamenejme, že k tomu, aby se tyto dva přepínače chovaly jako skupina, tj. aby „zapnutí“ jednoho znamenalo „vypnutí“ ostatních, stačí, že jsme je umístili do stejné komponenty RadioGroup.

Další úpravy jsou již poměrně snadné: Do třídy THlavniOkno přidáme dvě složky, barbva_mnoziny a barva_okoli, které inicializujeme počáteční barevnou kombinací. V handleru mn_nastaveniClick() pak otestujeme, která z možností byla vybrána (vlastnost Checked), a podle toho do těchto složek uložíme odpovídající barvy:

```
if(dlg_nastaveni->rb_cc->Checked)
{
  barva_mnoziny = InitBMno;     // InitBMno atd. jsou konstanty definované
  barva_okoli = InitBPoz;       // v souboru init.h
}
else
{
  barva_mnoziny = BMno2;
  barva_okoli = BPoz2;
}
```

Dvojici barev pak předáme funkci Vypocet() jako dodatečné parametry.

Poznámka

Všimněte si, že se stavový řádek chová, jako kdyby byl součástí obrázku, tj. posunuje se s ním. Pokud je obrázek větší něž klientská plocha okna, není ho vidět, pokud ho „nevytáhneme“ pomocí posuvníků.

Posuvníky

Program nyní může pracovat s bitovou mapou větší, než je klientská oblast okna. Proto je vhodné přidat do něj posuvníky. To ale už znáte ze 4. kapitoly, a proto to přenechám vám jako snadné cvičení. Ostatně řešení najdete jako obvykle na doprovodném CD, tentokrát v adresáři KAP06\021.

Pořadí komponent

Mezi řídicími prvky v dialogovém okně můžeme přecházet pomocí klávesy TAB, resp. v opačném pořadí pomocí kombinace SHIFT+TAB. Rozumné chování dialogového okna při procházení pomocí tabulátoru je také vizitkou programátora, který aplikaci dělal.

Pomocí vlastnosti TabStop můžeme určit, zda se má daná komponenta chovat jako „zarážka“ tabulátoru, tj. zda se na ni má přenést fokus. Pomocí vlastnosti TabOrder můžeme předepsat pořadí komponent. Toto pořadí má ale smysl pouze v rámci jedné karty, neboť samotným tabulátorem nelze přejít na další kartu. (Mezi kartami přecházíme pomocí klávesové kombinace CTRL+TAB.)

Výběr myší

Náš program má sloužit ke studiu fraktálů, a fraktály jsou množiny, které se skládají z částí podobných celku. Lze tedy očekávat, že uživatel začne zadáním nějaké větší oblasti a z ní si bude podle obrázku vybírat detaily, které bude chtít studovat podrobněji. Zadávání souřadnic v okně Nastavení mu to sice umožňuje, ale nepříliš pohodlně. Proto k programu přidáme možnost zvolit si novou oblast myší. Vybíranou oblast budeme označovat čárkovaným obdélníkem. Postup bude podobný jako při kreslení myší ve 4. kapitole. Využijeme k tomu událostí OnMouseDown, OnMouseMove a OnMouseUp.

Při stisknutí tlačítka myši, tj. při události OnMouseDown, nejprve ověříme, zda bylo stisknuto levé tlačítko myši. Pokud ano, nastavíme příznak, že bylo stisknuto, a zapamatujeme si souřadnice kurzoru. To bude jeden horní roh obdélníka, který bude v obrázku označovat vybranou oblast.

Při pohybu myši, tj. při události OnMouseMove, nejprve zkontrolujeme, zda je nastaven příznak stisknutí tlačítka. Pak se podíváme, zda už byl nějaký obdélník označující výběr nakreslen, a pokud ano, smažeme ho. Nakonec si zapamatujeme aktuální polohu kurzoru myši, nakreslíme nový obdélník (určený bodem, kde jsme tlačítko myši stiskli, a aktuálním bodem) a nastavíme příznak, že je nějaký obdélník nakreslen.

Při uvolnění tlačítka myši zkontrolujeme, zda byl nakreslen nějaký obdélník, a pokud ano, smažeme jej. K tomu využijeme kreslení v režimu pmXor, se kterým jsme se setkali už ve 3. kapitole. Pak zavoláme handler, který ošetřuje položku Nastavení z hlavního menu, a do editačních polí tohoto okna přeneseme zapamatované hodnoty. Po vybrání oblasti se tedy objeví dialogové okno Nastavení s novými hodnotami; pokud se nám nebudou líbit, stiskneme Storno a zamítneme je, jinak je použijeme při příštím kreslení.

To znamená, že musíme definovat dvě pomocné proměnné pro uložení příznaků, čtyři pro uložení souřadnic vrcholů obdélníka a handlery pro uvedené události. Vedle toho musíme lehce upravit handler volaný při události OnShow dialogového okna Nastavení.

Podívejme se nejprve na nové složky třídy THlavniOkno:

```
class THlavniOkno : public TForm
{
__published:       // IDE-managed Components
  // Beze změny, takže zde vynecháme
private:                  // User declarations
  // Jen nové složky:
  bool dole;                          // Je stisknuta myš?
  bool obdelnik;                      // Je nakreslen obdélník?
  int xlhv, ylhv, xpdv, ypdv;         // Souřadnice LH a PD rohu vybraného obdélnía
  void NakresliObdelnik();      // Kreslení obdélníka
public:                   // User declarations
  // Uvedeme zde jen nové složky
  bool Obdelnik() { return obdelnik;}   // Přístupové funkce
  int Xlhv() { return xlhv; }
  int Xpdv() { return xpdv; }
  int Ylhv() { return ylhv; }
  int Ypdv() { return ypdv; }
};
```

Nové složky ve třídě THlavniOkno (soubor okno1.h)

```
void __fastcall THlavniOkno::im_obrMouseDown(TObject *Sender,
        TMouseButton Button, TShiftState Shift, int X, int Y)
{
  if(Button == mbLeft)      // Je to levé tlačítko myši?
  {
    dole = true;                  // Nastav příznak
    xlhv = X;                     // a zapamatuj si souřadnice
    ylhv = Y;
  }
}

void __fastcall THlavniOkno::im_obrMouseMove(TObject *Sender,
                                                    TShiftState Shift, int X, int Y)
{
  if(dole)                            // Je-li stisknuto tlačítko
  {
    if(obdelnik)                // Je-li nakreslen obdélník, smaž ho
      NakresliObdelnik();
    obdelnik = true;            // Nastav příznak, že je nakreslen obdélník
    xpdv = X;                   // Zapamatuj si souřadnice druhého z vrcholů
    ypdv = Y;
    NakresliObdelnik();     // a nakresli obdélník
  }
}

void __fastcall THlavniOkno::im_obrMouseUp(TObject *Sender,
    TMouseButton Button, TShiftState Shift, int X, int Y)
{
  if(dole)                            // Je-li stisknuto tlačítko
  {
    dole = false;               // Zruš příznak, že je stisknuto tlačítko
    NakresliObdelnik();     // Smaž obdélník
    mn_nastaveniClick(Sender);      // Zavolej handler menu nastavení
    obdelnik = false;       // Zruš příznak, že je nakreslen obdélník
  }
}
```

Handlery, které se starají o výběr nové obladti myší (soubor okno1.cpp)

Kreslení obdélníka jsme odložili do zvláštní procedury. Důvod je jednoduchý: Před voláním metody TCanvas::Ellipse() musíme nastavit barvu pera, dále režim pmXor, který zaručí, že čáru smažeme prostě tím, že ji nakreslíme ještě jednou, a styl štětce, který zaručí, že se nakreslený obdélník nebude

vyplňovat. Po nakreslení obdélníka musíme tyto hodnoty obnovit, neboť jinak by jiné grafické operace nemusely fungovat správně!

```
void THlavniOkno::NakresliObdelnik()
{
 TCanvas* C = im_obr -> Canvas;        // Pomocná proměnná pro zjednodušení zápisu
 C -> Pen -> Color = clRed;            // Barva čáry
 TPenMode pm = C -> Pen -> Mode;       // Zapamatuj si mód a styl pera a styl štětce
 TPenStyle ps = C -> Pen -> Style;
 TBrushStyle bs = C -> Brush -> Style;
 C -> Pen -> Mode = pmXor;             // Pero: režim XOR
 C -> Pen -> Style = psDot;            // Čárkovaná čára
 C -> Brush -> Style = bsClear;        // Nevyplňovat obdélník
 C -> Rectangle(xlhv, ylhv, xpdv, ypdv);          // Nakresli obdélník
 C -> Pen -> Mode = pm;                // Obnov původní hodnoty
 C -> Pen -> Style = ps;
  C -> Brush -> Style = bs;
}
```

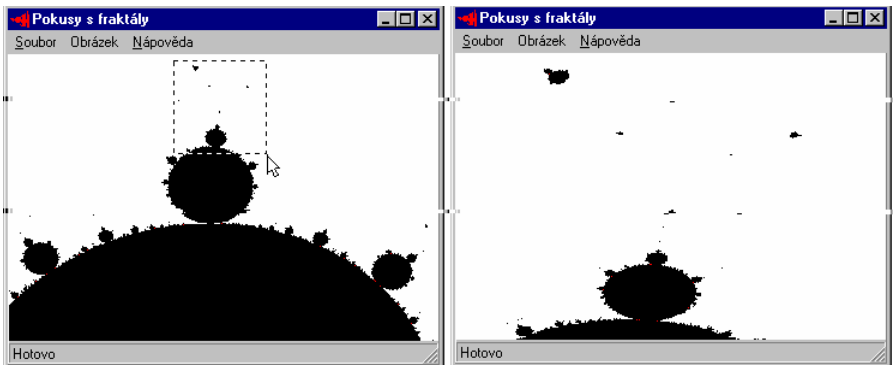
Kreslení obdélníka označujícího aktuální výběr (soubor okno1.cpp)

Poslední část úprav se týká handleru Tdlg_nastaveni::FormShow(), a to té jeho části, která se stará o „naplnění“ editačních polí hodnotami. Zde se nejprve podíváme, zda je vybrán obdélník, a pokud ano, vezmeme hodnoty z pomocných proměnných xlhv, ylhv a dalších, jinak použijeme stejně jako předtím hodnoty uložené v XLH, YLH atd.

```
void __fastcall Tdlg_nastaveni::FormShow(TObject *Sender)
{
  ed_xpd->Text = HlavniOkno->Obdelnik() ?
                 HlavniOkno->X(HlavniOkno->Xpdv()) : HlavniOkno->xpd();
  ed_xlh->Text = HlavniOkno->Obdelnik() ?
                 HlavniOkno->X(HlavniOkno->Xlhv()) : HlavniOkno->xlh();
  ed_ylh->Text = HlavniOkno->Obdelnik() ?
                 HlavniOkno->Y(HlavniOkno->Ylhv()) : HlavniOkno->ylh();
  ed_ypd->Text = HlavniOkno->Obdelnik() ?
                 HlavniOkno->Y(HlavniOkno->Ypdv()) : HlavniOkno->ypd();
  // ...a dále stejně jako v minulé verzi
}
```

Upravený handler pro zobrazení okna Nastavení (soubor nastav.cpp)

Výsledek ukazuje obrázek 6.12.



Obr. 6.12 Vybereme si myší detail a nakreslíme ho

6.4 Verze 3.0: Multithreading

Dvaatřicetibitové operační systémy na PC podporují multihtreading. To je způsob provádění programu, při kterém několik funkcí běží zároveň jako paralelní *vlákna*[36] (*thready*) výpočtu. To může usnadnit programování některých druhů úloh; jedním z poměrně obvyklých použití je rozdělení programu na dvě vlákna, z nichž jedno — hlavní — se stará o uživatelské rozhraní a druhé obstará

[36] Používá se také označení *tok výpočtu*. Ve starší literatuře se v této souvislosti zpravidla hovořilo o paralelních procesech, nicméně v terminologii 32bitových Windows se pod označením *proces* rozumí *program*.

složitý výpočet. Protože vlákna běží zároveň, může program zároveň[37] počítat a přitom reagovat na příkazy uživatele.

Než začneme svůj program upravovat, aby mohl používat více vláken, povíme si alespoň základní informace. Podrobné povídání o multithreadingu najdete v knize J. Richtera [6].

Problémy

Nic není zadarmo. Nové možnosti, které nám multithreading nabízí, jsou vykoupeny také novými problémy. Prvním (a většinou nejméně významným) je lehké zpomalení běhu programu způsobené administrativou při přepínání mezi vlákny.

Druhý problém je podstatně závažnější; týká se přístupu k prostředkům operačního systému. Zkuste si např. představit, jak by to dopadlo, kdyby dvě vlákna zároveň posílala data na tiskárnu, tiskla zároveň dva různé výstupy.

Neméně nepříjemné komplikace mohou nastat i při přístupu k proměnným a jiným datům, které jsou společné pro celý program. Představme si následující situaci. Ve vlákně A napíšeme příkaz

```
Uprav(&i);          // i je glob ln  prom nn
```

a ve vlákně B napíšeme

```
Pou ij(i);          // i je t   glob ln  prom nn
```

Ve vlákně A vstoupí výpočet do funkce Uprav() a začne měnit hodnotu globální proměnné i. V tom okamžiku operační systém vlákno A přeruší a předá řízení vláknu B, které hodnotu proměnné použije. Ale co použije? Hodnotu před úpravou, po ní, nebo nějaký mezivýsledek, který nemá žádný význam?

Je asi jasné, že podobné problémy mohou nastat i na počítačích s více procesory, kde vlákna mohou běžet opravdu paralelně.

Synchronizace

Z předchozího povídání plyne, že některé operace — přístup k datům, k prostředkům operačního systému apod. — je třeba synchronizovat. Vedle toho se může stát, že potřebujeme, aby jedno vlákno počkalo na dokončení výpočtu ve druhém vláknu.

Základní prostředky pro synchronizaci přístupu k datům jsou semafory, kritické sekce a mutexy. Všimneme si blíže kritických sekcí. Chceme-li zabránit současnému přístupu několika vláken k určité proměnné, uzavřeme veškeré operace s ní do kritické sekce. Vlákno, které ji chce použít, musí nejprve do této kritické sekce vstoupit. Pak provede příslušné operace a z kritické sekce vystoupí.

Přitom platí, že do kritické sekce smí vstoupit vždy jen jedno vlákno. Pokud je kritická sekce už obsazena, musí počkat — operační systém mu nedovolí pokračovat, dokud nebude kritická sekce volná.

Mutex funguje podobně, slouží však především k synchronizaci mezi procesy. (Kritické sekce existují v programu, mutex existuje mimo program v operačním systému.)

Ovšem takováto synchronizace s sebou přináší další problémy. Podívejme se na případ, kdy dvě vlákna, A a B, používají proměnné x a y. Každá s těchto proměnných je uzavřena do jedné kritické sekce, řekněme ksx a ksy. Co když vlákno A vstoupí do kritické sekce ksx a pak se pokusí vstoupit do kritické sekce ksy, aniž opustí ksx, *a zároveň* vlákno B vstoupí do kritické sekce ksy a pak se pokusí vstoupit do kritické sekce ksx?

Výsledek bude, že vlákno A vstoupí do kritické sekce ksx, ale do ksy už ne, protože v té je B. Na druhé straně B vstoupí do xsy, ale do ksx už ne, protože v té je A. Obě vlákna budou čekat, až jim to druhé uvolní kritickou sekci, a program „zamrzne“. (Této situaci se říká *dead lock*.) Z toho plyne, že pokud budeme v programu používat několika kritických sekcí zároveň, musíme si dát pozor, abychom do nich vstupovali vždy ve stejném pořadí. V předchozím příkladu by např. stačilo, kdybychom v obou vláknech vstupovali nejprve do ksx a pak do ksy.

Pro komunikaci vedlejších vláken s hlavním vláknem, které se stará o uživatelské rozhraní, se zpravidla používá zpráv Windows.

[37] Neberte to příliš doslova. Pokud máte počítač s jediným procesorem, poběží jednotlivá vlákna střídavě. Ovšem intervaly, ve kterých se budou střídat, jsou natolik krátké, že se nám to bude jevit, jako kdyby opravdu běžela zároveň.

Poznamenejme, že klasická knihovna jazyka C není napsána tak, aby ji bylo možno používat ve vícevláknovém prostředí. Proto je součástí překladačů pro Win32 knihovna speciálně upravená.

Vlastnosti vláken

Každé vlákno má svou prioritu. Ta určuje, jak mu bude operační systém přidělovat čas. Později se seznámíme s předdefinovanými konstantami, které ji popisují, zatím si jen řekneme, že většinou vystačíme s tzv. normální prioritou. Dáme-li vlákny vyšší prioritu než mají ostatní, nebude operační systém jiným vláknům přidělovat čas, dokud toto vlákno neskončí nebo nepozastaví svoji činnost.

Běžící vlákno lze pozastavit (suspendovat) nebo ukončit. Pozastavené vlákno lze opět spustit a bude pokračovat z místa, kde přestalo.

Jestliže v jednom vláknu zavoláme nějakou funkci, poběží v rámci tohoto vlákna. Nic — kromě výše zmíněných problémů se synchronizací — nám nebrání volat jednu funkci z několika vláken. Protože každé vlákno má svůj zásobník, poběží v několika instancích nezávisle. (Problémy mohou nastat, použijeme-li v takové funkci lokální statickou proměnnou. Musíme si dát pozor, aby funkce volané z threadů byly reentrantní.)

Multithreading v C++ Builderu

C++ Builder obsahuje třídu TThread, která zapouzdřuje nástroje pro práci s vlákny z Windows API. Tato třída slouží jako společný předek vláken používaných v programu. Chceme-li použít ve svém programu více vláken, definujeme si potřebné třídy jako potomky typu TThread. Vlastní výpočet ve vláknu bude mít na starosti virtuální metoda Execute(), kterou musí uživatel předefinovat. Skončí-li tato metoda, skončí běh vlákna.

Konstruktor této třídy má parametr CreateSuspended typu bool, který určuje, zda se vlákno vytvoří suspendované nebo zda se rovnou rozeběhne. V konstruktoru odvozené třídy také zpravidla nastavujeme prioritu (vlastnost Priority) a chování při ukončení (vlastnost FreeOnTerminate). Implicitně je FreeOnTerminate == false, tj. komponenta představující vlákno při ukončení neuvolní, musíme se o to postarat sami. Poznamenejme, že konstruktor (a také destruktor) vlákna běží v rámci vlákna, které ho volá, nikoli v rámci nového vlákna.

Chceme-li vlákno pozastavit, použijeme k tomu metodu Suspend(). Chceme-li, aby pokračovalo, zavoláme metodu Resume(). Chceme-li zjistit stav vlákna, použijeme vlastnost Suspended.

Chceme-li běh vlákna předčasně ukončit, zavoláme metodu Terminate(), která nastaví na true vlastnost Terminated — příznak, který říká, že má vlákno skončit. Metoda Execute() by měla v pravidelných intervalech kontrolovat, zda náhodou není nastaven tento příznak nastaven. (Aplikační rozhraní Win32 obsahuje také funkce, které mohou vlákno ukončit „násilně“, např. TerminateThread(). Používáme je ale jen výjimečně.)

Při ukončení vlákna nastane událost OnTerminate. Handler, který na ni reaguje, může provést „úklid“ po vláknu. (Připomeňme si, že události jsou implementovány jako ukazatele na metody. Stačí tedy napsat metodu s odpovídajícím prototypem a přiřadit její adresu proměnné OnTerminate.) Důležité je, že tento handler běží v rámci hlavního vlákna, takže se zde nemusíme starat o synchronizaci.

Potřebujeme-li na počkat, než nějaké vlákno skončí, použijeme jeho metodu WaitFor(). Tato metoda skončí až poté, co dané vlákno skončí, a vrátí hodnotu, kterou jsme přiřadili vlastnosti ReturnValue daného vlákna.

V C++ Builderu se obvykle hlavní vlákno stará o uživatelské rozhraní a v případě potřeby si spouští další pomocná vlákna, která obstarají složité výpočty na pozadí. Přitom by ale tato vlákna neměla pracovat s prvky uživatelského rozhraní. Veškeré operace s komponentami v uživatelském rozhraní by mělo obstarat vlákno, které se o ně stará; pokud to není možné, použijeme metodu Synchronize(), které jako parametr dáme ukazatel na metodu bez parametrů, jež provede potřebnou operaci. Výjimku tvoří komponenta TCanvas, která obsahuje metody Lock() a Unlock(), umožňující synchronizaci.

Kritické sekce jsou v C++ Builderu implementovány ve třídě TCriticalSection. Pro vstup do kritické sekce slouží metoda Enter(), pro výstup z ní Leave().

Program

Upravíme tedy předchozí verzi programu na kreslení fraktálů tak, aby výpočet probíhal v samostatném vláknu.

Nové vlákno

Začneme tím, že k programu připojíme nové vlákno. Použijeme k tomu příkaz File | New... a v okně zásobníku objektů zvolíme položku Thread Object. Objeví se dialogové okno, ve kterém zadáme jméno třídy nového vlákna; zvolíme TVypocet.

Prostředí vytvoří hlavičkový a zdrojový soubor s deklarací třídy; uložíme je pod jménem tvypoc.h a tvypoc.cpp. Nově vytvořená třída obsahuje konstruktor a virtuální metodu Execute().

Při výpočtu budeme potřebovat hodnoty uložené ve třídě hlavního okna. Abychom se vyhnuli problémům se synchronizací, okopírujeme si je při vytvoření vlákna do tohoto objektu. Přeneseme sem také kopie funkcí X() a Y(). Dále do této třídy přidáme metodu Konec(), která bude sloužit jako handler pro událost OnTerminate, a funkci NastavStavovouRadku(), o které budeme hovořit později. Definici třídy TVypocet ukazuje následující výpis.

```
class TVypocet : public TThread
{
private:                                  // Data okopírovaná z hlavního okna:
  double XLH, YLH, XPD, YPD;      // Souřadnice LH a PD rohu (v rovině X-Y)
  int sirka, vyska;                  // Rozměry bitové mapy v pixelech
  double ax, ay;                        // Konstanty pro přepocet souřadnic na pixely
  TColor barva_mnoziny, barva_okoli;   // Barvy pro kreslení
  int pocetIteraci;
  TCanvas * C;                          // Pro usnadnění zápisu
  AnsiString TextStavoveRadky; // Text do stavové řádky
  void __fastcall NastavStavovouRadku();          // Zápis do staové řádky
protected:
  void __fastcall Execute();       // Vlastní výpočet
public:
  __fastcall TVypocet(bool CreateSuspended);
  double X(int x) { return ax*x+XLH; }                 // Přepočet souřadnic
  double Y(int y) { return ay*y+YLH; }
  void __fastcall Konec(TObject * Sender);   // Handler pro OnTerminate
};
```

Definice třídy Tvypocet (soubor thvypoc.h)

Konstruktor se postará především o inicializace proměnných. Vedle toho nastaví normální prioritu (přiřadí hodnotu tpNormal vlastnosti Priority), určí, že po ukončení výpočtu se má automaticky uvolnit paměť vlákna (přiřadí true vlastnosti FreeOnTerminate) a definuje handler pro událost OnTerminate.

```
__fastcall TVypocet::TVypocet(bool CreateSuspended)
 : TThread(CreateSuspended), barva_mnoziny(HlavniOkno->bmnoziny()),
   barva_okoli(HlavniOkno->bokoli()), XLH(HlavniOkno->xlh()),
   YLH(HlavniOkno->ylh()), XPD(HlavniOkno->xpd()), YPD(HlavniOkno->ypd()),
   sirka(HlavniOkno->Sirka()), vyska(HlavniOkno->Vyska()), ax(HlavniOkno->Ax()),
    ay(HlavniOkno->Ay()), C(HlavniOkno->im_obr->Canvas),
    pocetIteraci(HlavniOkno->PocetIteraci())
{
  Priority = tpNormal;
  FreeOnTerminate = true;
  OnTerminate = &Konec;
}
```

Konstruktor našeho vlákna (soubor thvypoc.cpp)

Vlastní výpočet v metodě Execute() bude jednoduchý. V cyklu budeme pro jednotlivé pixely volat funkci Vypocet() a vypočtenou barvu přiřadíme odpovídajícímu prvku pole Pixels. Abychom nemuseli stále znovu opisovat konstrukci HlavniOkno->im_obr->Canvas, zavedli jsme si zkratku C; je to ukazatel na TCanvas, který jsme inicializovali v konstruktoru vlákna.

V cyklu obsahujícím výpočet barvy také kontrolujeme hodnotu vlastnosti Terminated, a pokud obsahuje hodnotu true, skončíme.

Pro výpis postupu výpočtu do stavové řádky použijeme metodu NastavStavovouRadku(), kterou budeme volat prostřednictvím metody Synchronize(), neboť pracuje se součástí uživatelského rozhraní a to má na starosti hlavní vlákno. (Metoda Synchronize() však vyžaduje ukazatel na metodu bez parametrů, a proto si musíme vystupující text uložit do pomocné proměnné TextStavoveRadky.)

```
void __fastcall TVypocet::Execute()
{
  for(int ix = 0; ix < sirka; ix++)
  {
   for(int iy = 0; iy < vyska && !Terminated; iy++)
   {
     TColor barva = Vypocet(
                X(ix), Y(iy), pocetIteraci, barva_mnoziny, barva_okoli);
     C->Lock();
     C -> Pixels[ix][iy] = barva;
     C-> Unlock();
   }
   TextStavoveRadky = AnsiString(100*ix/sirka)+"%";
   Synchronize(NastavStavovouRadku); // Nutno synchronizovat!
  }
}
```

Metoda Execute() představuje „tělo“ vlákna (soubor thvypoc.cpp)

Metoda NastavStavovouRadku() opravdu udělá to, co říká její jméno. Musí ale být typu void __fastcall a bez parametrů.

```
void __fastcall TVypocet::NastavStavovouRadku()
{
  HlavniOkno->sb_stav->SimpleText = TextStavoveRadky;
}
```

Třída hlavního okna

Nyní se ve stručnosti podíváme na úpravy, které je třeba provést ve třídě THlavniOkno. Budou se skládat ze dvou kroků: Nejprve do deklarace této třídy přidáme ukazatel na nové vlákno a pak upravíme metody, které se starají o výpočet — tedy reakce na příkazy Výpočet a Přeruš. Většinu problémů se vzájemným vztahem výpočtového vlákna a hlavního vlákna, které se stará o uživatelské rozhraní, vyřešíme tím, že po dobu výpočtu zakážeme příkazy Nový..., Ulož... a Výpočet a v době, kdy výpočet neběží, příkazy Přeruš a Ukonči.

Nejprve tedy do deklarace v souboru okno1.h přidáme ukazatel na vlákno, soukromou složku

```
TVypocet* vyp;
```

V konstruktoru jej inicializujeme hodnotou 0. (Konstruktor by měl inicializovat všechny složky nějakou rozumnou hodnotou.)

Pak přejdeme do vizuálního návrhu a hlavního okna a do nabídky Obrázek přidáme podřízenou položku Ukonči, která bude mít za úkol předčasně ukončit výpočet. (V předchozí verzi jsme příkazem Přeruš ve skutečnosti výpočet předčasně ukončili. Nyní budeme mít možnost po přerušení výpočtu pokračovat z místa, kde jsme skončili.) Nabídka Obrázek tedy nyní obsahuje příkazy Nastavení, Výpočet, Přeruš a Ukonči.

Výpočet

Začneme u metody, která reaguje na příkaz Výpočet. Ta se postará o úvodní úpravy nabídky, vytvoří nové vlákno a ihned ho spustí (předá konstruktoru vlákna parametr false).

```
void __fastcall THlavniOkno::mn_vypocetClick(TObject *Sender)
{
  ulozeno = false;                  // Obrázek ještě nebyl uložen
  mn_prerus ->Enabled = true;       // Uprav nabídky
  mn_ukonci ->Enabled = true;
  mn_vypocet->Enabled = false;
  mn_novy   ->Enabled = false;
  mn_uloz   ->Enabled = false;
  vyp = new TVypocet(false);        // Vytvoř výpočtové vlákno a spusť ho
}
```

Spuštění výpočtu (soubor okno1.cpp)

Tato metoda se nemůže postarat o vrácení nabídky do původního stavu. Kdybychom napsali

```
void __fastcall THlavniOkno::mn_vypocetClick(TObject *Sender) // CHYBN   E EN
{
  ulozeno = false;                  // Obrázek ještě nebyl uložen
  mn_prerus ->Enabled = true;       // Uprav nabídky
  // ... a další úpravy
```

```
  vyp = new TVypocet(false);          // Vytvoř výpočtové vlákno a spusť ho
  mn_prerus ->Enabled = false;  // Vrať nabídky do původního stavu (!)
  // ... a další úpravy ...
}
```

úpravy nabídek by se vůbec neprojevily. Po vytvoření a spuštění výpočtového vlákna totiž běh hlavního vlákna pokračuje, takže bychom tyto úpravy ihned po spuštění výpočtu zase zrušili.

Nepomůže nám ani metoda WaitFor(). Kdybychom napsali

```
void __fastcall THlavniOkno::mn_vypocetClick(TObject *Sender)// CHYBN    E EN
{
  ulozeno = false;                    // Obrázek ještě nebyl uložen
  mn_prerus ->Enabled = true;         // Uprav nabídky
  // ... a další úpravy
  vyp = new TVypocet(false);          // Vytvoř výpočtové vlákno a spusť ho
  vyp -> WaitFor();                        // Počkej, až výpočet skončí (!!)
  mn_prerus ->Enabled = false;  // Vrať nabídky do původního stavu (!)
  // ... a další úpravy ...
}
```

Dostali bychom program, který se chová naprosto stejně jako verze 1.0. Voláním této metody totiž zastavíme běh hlavního vlákna, které bude čekat, až výpočtové vlákno skončí, takže nebude zpracovávat zprávy, nebude kreslit atd.

Nezbývá tedy než přesunout úpravy nabídek po skončení výpočtu jinam — a k tomu se hodí právě handler reagující na událost OnTerminate. Nazveme jej Konec(), a protože běží po skončení metody Execute(), tedy v rámci hlavního vlákna, nemusí se starat o synchronizaci.

```
void __fastcall TVypocet::Konec(TObject* Sender)
{
  HlavniOkno -> sb_stav   -> SimpleText = "Hotovo";
  HlavniOkno -> mn_prerus  -> Enabled = false;      // Závěrečné úpravy nabídky
  HlavniOkno -> mn_vypocet -> Enabled = true;
  HlavniOkno -> mn_novy    -> Enabled = true;
  HlavniOkno -> mn_uloz    -> Enabled = true;
  HlavniOkno -> mn_ukonci  -> Enabled = false;
}
```

Úklid po ukončení výpočtu (soubor thvypoc.cpp)

Nyní se podíváme na metodu, která reaguje na příkaz Přeruš. K dočasnému přerušení výpočtu použijeme metodu Suspend(). Ovšem pak je velice snadné pokračovat z místa, kde jsme výpočet přerušili — stačí zavolat metodu Resume(). Kvůli pokračování není nutno přidávat do nabídky novou položku; stačí, když po přerušení změníme nápis na položce Přeruš na Pokračuj. Aktuální stav vlákna — zda je suspendované nebo ne — zjistíme pomocí vlastnosti Suspended.

```
void __fastcall THlavniOkno::mn_prerusClick(TObject *Sender)
{
  if(vyp -> Suspended)                // Je-li vlákno suspendováno, chceme pokračovat
  {
   mn_prerus -> Caption = "Přeruš";       // Uprav nápis v nabídce
   vyp->Resume();
  }
  else                                          // jinak chceme výpočet přerušit
  {
   vyp -> Suspend();
   mn_prerus -> Caption = "Pokračuj";    // Uprav nápis v nabídce
   sb_stav -> SimpleText = "Přerušeno";
  }
}
```

Metoda reagující na příkaz k přerušení nebo k pokračování výpočtu (soubor okno1.cpp)

Také předčasné ukončení je jednoduché: Prostě zavoláme metodu Terminate(), která nastaví příznak Terminated. (Jak víme, metoda Execute() musí tento příznak neustále kontrolovat.) Jediný problém se týká ukončení přerušeného (suspendovaného) vlákna. To musíme nejprve spustit, aby mohlo reagovat na příkaz k přerušení.

```
void __fastcall THlavniOkno::mn_ukonciClick(TObject *Sender)
{
  if(vyp -> Suspended)        // Přerušené vlákno nejprve spusť
  {
   vyp -> Resume();
   mn_prerus -> Caption = "Přeruš";
  }
```

```
  vyp -> Terminate();      // Ukonči výpočet
}
```

Předčasné ukončení výpočtu (soubor okno1.cpp)

Program v této podobě najdete na doprovodném CD v adresáři KAP06\03.

Poznámka

Náš program při kreslení velmi nepříjemně poblikává. (Pokusně lze zjistit, že poblikávání zmizí, je-li rozměr obrázku stejný nebo větší než rozměr klientské oblasti kreslicího okna.) Nabízí se následující řešení, které blikání alespoň částečně omezí.

V programu si vytvoříme objekt třídy TBitmap, např.

```
Tbitmap * bmp;
```

a budeme kreslit do jeho plátna (canvasu). Výsledek pak jednou za čas, např. po vykreslení celé jedné přímky, přeneseme již nakreslenou část obrázku do okna pomocí metody Assign(), tj. příkazem

```
HlavniOkno -> im_obr -> Picture -> Bitmap -> Assign(bmp);
```

Tento příkaz je ale třeba také synchronizovat, tj. „zabalit“ do metody bez parametrů a tu volat prostřednictvím Synchronize(). Vedle toho se nabízí ještě několik dalších úprav, např. možnost zadávání „kroku“ (počtu svislých řad pixelů, po kterých budeme obrázek vykreslovat). Program v této podobě najdete na doprovodném CD v adresáři KAP06\031 jako verzi 3.1.

6.5 Verze 4.0: Funkce v dynamické knihovně

Naším dalším krokem bude umístění funkce, která počítá barevné hodnoty jednotlivých pixelů, do samostatné dynamické knihovny (DLL[38]). To s sebou přinese výhody i nevýhody, např.:

- ✧ Program tím rozdělíme na několik samostatně překládaných částí. To sice zpravidla znamená, že při jeho šíření musíme použít instalační program, ale na druhé straně to usnadní aktualizaci (stačí dodat novou verzi jen některých knihoven).
- ✧ Uživatel může k programu v případě potřeby doplnit svou vlastní dynamickou knihovnu (a tak např. studovat fraktál, o jehož existenci my jako autoři programu nemáme tušení).
- ✧ Z dynamických knihoven lze exportovat (tj. poskytnout k použití ostatním programům) pouze funkce vyhovující pravidlům jazyka C. Nelze z nich exportovat např. objektové typy nebo funkce přeložené podle pravidel jazyka C++. (Uvnitř knihoven — „pro vnitřní potřebu“ — lze objekty nebo funkce definované v C++ samostatně používat.)
- ✧ Dynamické knihovny mohou být napsány v podstatě v jakémkoli programovacím jazyce.

Projekt dynamické knihovny

Nejprve vytvoříme nový projekt. Příkazem File | New... vyvoláme zásobník objektů a v něm zvolíme položku DLL. Prostředí C++ Builderu vytvoří projekt a zdrojový soubor; uložíme je pod názvem vypocet. IDE vytvoří soubory vypocet.cpp a vypocet.bpr. V souboru vypocet.cpp najdeme funkci

```
int WINAPI DllEntryPoint(HINSTANCE hinst, unsigned long reason, void*)
{
 return 1;
}
```

Operační systém tuto funkci zavolá vždy při spuštění nebo ukončení programu nebo při zavedení této knihovny do paměti nebo při jejím uvolnění. Skutečnou příčinu volání lze určit podle hodnoty druhého parametru; podrobnosti najdete v nápovědě. Tuto funkci lze užít k inicializacím dat používaných v rámci knihovny. Pokud se inicializace podaří, vrátí 1, jinak vrátí 0. My tyto funkci nebudeme potřebovat, a proto ji ponecháme beze změny.

Za ni doplníme definici funkce Vypocet(), která počítá barvy jednotlivých bodů obrázku (a případně dalších pomocných funkcí, typů, proměnných atd.).[39] Deklaraci funkce Vypocet() musíme lehce upravit:

[38] Dynamické knihovny jsou knihovny, které se zavádějí do paměti, až když o to program požádá, a uvolňují z paměti v okamžiku, kdy program prohlásí, že je již nepotřebuje.

- ✧ Musíme ji definovat jako funkci napsanou v jazyce C. K tomu poslouží specifikace extern "C".
- ✧ Dále musíme specifikovat, že funkci Vypocet() chceme z knihovny exportovat (tj. že ji chceme používat i mimo knihovnu). K tomu nám poslouží nestandardní modifikátor __declspec(dllexport). (Ve starších překladačích se používal modifikátor __export. Protože ale klíčové slovo export je nyní součástí standardu jazyka C++ a má jiný význam, používá se v novějších překladačích zápis pomocí __declspec(). Toto klíčové slovo se používá i v dalších konstrukcích.)

Prototyp této funkce tedy bude

```
extern "C" __declspec(dllexport)
TColor Vypocet(double x, double y, int n, TColor bmn, TColor bok);
```

Zároveň vytvoříme hlavičkový soubor vypocet.h, který bude obsahovat tento prototyp. To je vše. Nyní náš projekt přeložíme. Všimněte si, že C++ Builder vytvořil vedle souboru vypocet.dll také soubor vypocet.lib. To je tzv. importní knihovna, o které si brzy povíme více.

Použití dynamické knihovny

C++ Builder nenabízí zvláštní prostředky pro použití dynamických knihoven. Mysíme tedy využít možností, které nám poskytuje Windows API. Funkce z dynamických knihoven lze používat dvěma způsoby.

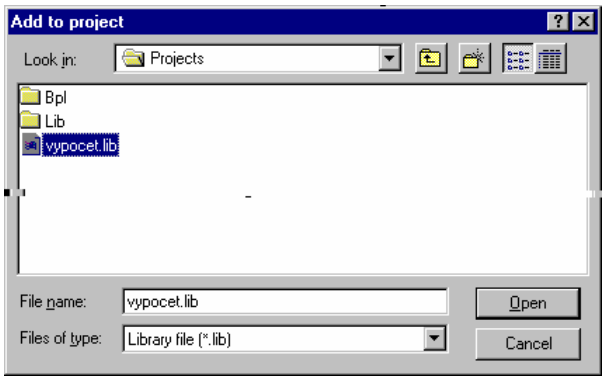
- ✧ Stejně jako jakékoli jiné funkce. Do svého programu vložíme odpovídající hlavičkový soubor a funkci voláme jejím identifikátorem. Při sestavování musíme k programu připojit odpovídající importní knihovnu. V tomto případě se dynamická knihovna zavede do paměti hned při spuštění programu a uvolní se při jeho ukončení.
- ✧ Knihovnu zavedeme do paměti voláním funkce LoadLibrary() až ve chvíli, kdy ji potřebujeme. Pomocí funkce GetProcAddress() pak získáme adresu funkce, kterou chceme volat.

První možnost

Začneme první možností, neboť ta bude vyžadovat nejmenší množství změn v projektu. Nejprve z projektu fraktálového programu odstraníme modul vypoc, neboť ten už nebudeme potřebovat. Vyvoláme si okno správce projektů (CTRL+ALT+F11), v něm vybereme modul vypoc a stiskneme klávesu DEL.

Pak prohlédneme hlavičkové soubory a nahradíme v nich direktivu #include "vypoc.h" direktivou #include "vypocet.h".

Nakonec příkazem Project | Add to project otevřeme dialogové okno Add to project, v něm jako typ souboru zvolíme knihovnu (Library file) a zadáme importní knihovnu vypocet.lib (obr. 6.13). Poté program přeložíme a spustíme.



Obr. 6.13 Připojíme k projektu importní knihovnu

Při tomto postupu je velice důležitá importní knihovna vypocet.lib. Ta totiž zajistí, že

- ✧ linker program dokáže sestavit (najde funkci Vypocet(), a to právě v importní knihovně),

[39] Ve skutečnosti nám nic nebrání vložit za funkci DllEntryPoint() lehce upravený obsah souboru vypoc.cpp. Úpravy se týkají především prototypu funkce Vypocet().

- ✧ po spuštění programu se dynamická knihovna vypocet.dll zavede do paměti a na konci se uvolní,
- ✧ při volání funkce Vypocet() se jí předají správně parametry.

Představuje tedy jakýsi spojovací článek mezi programem a dynamickou knihovnou. (Poznamenejme, že podobným způsobem se volají funkce z Windows API.)

Na doprovodném CD v adresáři KAP06\04 najdete jednak projekt fra4 programu pro kreslení fraktálů a jednak projekt vypocet pro dynamickou knihovnu obsahující funkci pro výpočet barvy jednotlivých bodů.

Druhá možnost

Nyní se podíváme, jak se náš program změní, jestliže se rozhodneme zavádět dynamickou knihovnu do paměti „ručně“. Při tomto postupu nepotřebujeme hlavičkový soubor, neboť funkci pro výpočet fraktálu budeme volat prostřednictvím ukazatele. Nepotřebujeme také importní knihovnu.

Před použitím potřebujeme znát jméno souboru s dynamickou knihovnou a jméno funkce, kterou budeme volat (obojí jako znakové řetězce).

V našem případě si nejprve deklarujeme typ ukazatele na funkci pro výpočet fraktálu a proměnnou tohoto typu

```
typedef TColor (*fun)(double, double, int, TColor, TColor);
fun f;
```

Pak zavedeme dynamickou knihovnu do paměti:

```
HINSTANCE h = LoadLibrary("vypocet.dll");
```

Funkce LoadLibrary() vrátí hodnotu typu HINSTANCE, což je v současné verzi Windows jen jiné jméno pro void*. Pokud se dynamickou knihovnu nepodaří zavést, vrátí 0, jinak vrátí nenulovou hodnotu.

Pak z knihovny získáme pomocí GetProcAddress() adresu požadované funkce.

```
f = (fun)GetProcAddress(h, "_Vypocet");
```

Pokud tato funkce neuspěje, vrátí 0, jinak vrátí ukazatel, který můžeme používat běžným způsobem, např.

```
bmp -> Canvas -> Pixels[ix][iy] = f(X(ix), Y(iy), pocIt, bmn, bok);
```

Po skončení výpočtu dynamickou knihovnu uvolníme příkazem

```
FreeLibrary(h);
```

Poznámky

- ✧ Řetězec předaný funkci LoadLibrary() může obsahovat i cestu. Pokud ji neobsahuje, hledá systém dynamickou knihovnu nejprve v domovském adresáři aplikace, pak v aktuálním adresáři, pak v podadresáři system domovského adresáře Windows (ve Windows NT nejprve v system32, pak v system), pak v domovském adresáři Windows a nakonec v adresářích uvedených v systémové proměnné PATH.
- ✧ Pokud řetězec udávající jméno neobsahuje příponu, předpokládá se .DLL. Pokud jméno končí tečkou, znamená to, že nemá příponu.
- ✧ Všimněte si znaku podtržení před jménem funkce Vypocet ve volání funkce GetProcAddress(). Ke jménům všech funkcí přeložených podle konvencí jazyka C překladač připojí úvodní podtržítko a my ho zde musíme uvádět.
- ✧ Funkce lze v dynamických knihovnách vyhledávat také podle pořadových čísel, např.

```
f = (fun)GetProcAddress(h, (LPCSTR)1);
```

- ✧ Hodnotu vrácenou funkcí GetProcAddress() je v C++ třeba přetypovat.
- ✧ Na doprovodném CD najdete v adresáři KAP06\041 verzi 4.1 našeho programu, která umožňuje zvolit si za běhu programu dynamickou knihovnu obsahující funkci pro výpočet fraktálu. Jméno dynamické knihovny je uloženo v proměnné typu AnsiString a využívá ji metoda TVypocet::Execute() v modulu thvypoc. Přiloženy jsou i dvě dynamické knihovny pro výpočet dvou různých obrázků.

7. První kroky v databázích

V této kapitole začneme vytvářet databázové aplikace v C++ Builderu. Budeme přitom především využívat vzorových databází, které jsou součástí instalace C++ Builderu.

7.1 První příklad: prohlížeč databáze

Než se pustíme do výkladu o databázové architektuře C++ Builderu a o databázových komponentách, které v něm máme k disposici, ukážeme si příklad: Napíšeme program, který nám umožní prohlížet si databázi zemí na obou amerických kontinentech. Tabulka[40] těchto zemí je součástí instalace C++ Builderu a najdete ji v adresáři Program Files\Borland\Borland Shared\Data pod jménem country.db. Je to tedy tabulka pro Paradox; postup by ale byl prakticky stejný, i kdyby se jednalo o tabulku pro jinou stolní databázi nebo o tabulku na databázovém serveru[41].

Pro tabulky v tomto adresáři je vytvořen *alias* BCDEMOS. Později si vysvětlíme, co to znamená.

Nyní se již můžeme pustit do programování. Začneme nový projekt, který uložíme pod jménem db1. Formulář aplikace pojmenujeme tradičně HlavniOkno a odpovídající modul pojmenujeme okno1.

Pak najdeme na paletě Data Access komponentu Table a vložíme ji do formuláře — tedy do vizuálního návrhu naší budoucí aplikace; protože jde o nevizuální komponentu, je jedno, kam ji položíme. Pojmenujeme ji např. tabulka (vlastnost Name).

Na komponentu Table se můžeme dívat jako na programový model databázové tabulky. Bude se starat o veškerý styk s tabulkou, musíme jí ale říci, se kterou. Vyhledáme tedy v inspektoru objektů vlastnost DatabaseName, která bude obsahovat alias zobrazovaných dat, a zadáme BCDEMOS. (U této vlastnosti nám inspektor objektů nabídne rozbalovací seznam dostupných aliasů, ze kterého si můžeme vybrat. Pokud pracujeme se stolní databází, můžeme sem místo aliasu zapsat cestu do adresáře, kde je tabulka.)

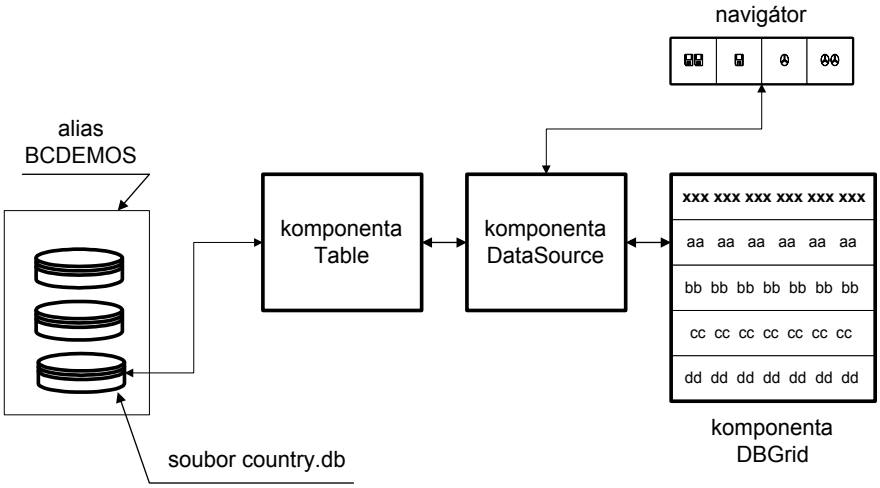
Nyní ještě musíme určit, kterou z tabulek určených daným aliasem vlastně chceme. K tomu použijeme vlastnost TableName. I zde si můžeme vybrat z rozbalovacího seznamu; zvolíme coubntry.db.

Dále vložíme do vizuálního návrhu komponentu DataSource, kterou najdeme také na paletě Data Access. Pojmenujeme ji zdroj a její vlastnosti DataSet dáme hodnotu tabulka. Tím říkáme, odkud má čerpat data. Tato komponenta bude přenášet data z tabulky do komponent, které je budou zobrazovat.

Nakonec vložíme do vizuálního návrhu komponentu DBGrid, která představuje „mřížku“ zobrazující jednotlivé záznamy z databáze, a komponentu BDNavigator, která obsahuje tlačítka pro procházení tabulkou, přidávání a mazání záznamů. Obě tyto komponenty najdeme na paletě Data Controls. Klidně jim ponecháme nabídnutá jména a u obou přiřadíme v inspektoru objektů vlastnosti DataSource hodnotu zdroj. Tím říkáme, že mají čerpat data z této komponenty. Vzájemné propojení komponent naznačuje obr. 7.1.

[40] K pojmu *tabulka* se ještě vrátíme. Zatím stačí, když vyjdeme z představy, že to je soubor, který obsahuje informace uspořádané do *záznamů*.

[41] Pokud budete chtít použít tabulky na lokálním databázovém serveru InterBase, použijte alias IBLocal. Při zadávání jména tabulky budete požádáni o uživatelské jméno a heslo; implicitně je nastaveno jméno SYSDBA a heslo masterkey. Pomocí utilit pro správu tohoto serveru si můžete zavést vlastní uživatelské jméno a heslo.

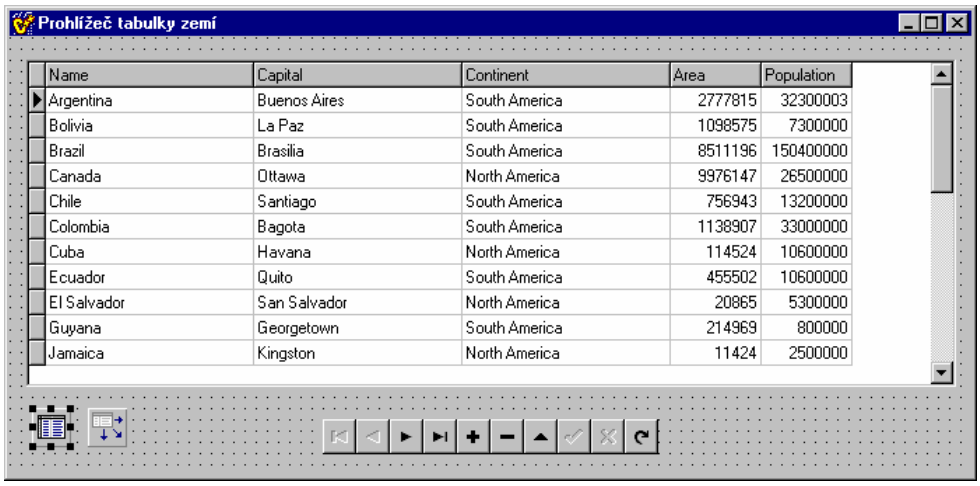


Obr. 7.1 Propojení komponent a tok dat v prvním databázovém programu

Nyní se vrátíme ke komponentě tabulka. Vyhledáme v inspektoru objektů vlastnost Active a dáme jí hodnotu true, čímž tabulku opravdu aktivujeme, tj. tím přikážeme komponentě Table navázat spojení a získat od ní data. Tato data se ihned zobrazí v komponentě TDBGrid (obr. 7.2).

Tím jsme hotovi — program uložíme, přeložíme a spustíme. Bude zobrazovat tabulku zemí, ve které můžeme přecházet mezi jednotlivými záznamy (tj. řádky) pomocí prvních čtyř tlačítek navigátoru. Další dvě tlačítka (se znaménky + a -) umožňují přidat do databáze nový záznam nebo ho z ní odstranit. Tlačítko s trojúhelníčkem umožňuje editovat aktuální záznam. Přidání nového záznamu nebo změny v některém ze záznamů potvrdíme stisknutím tlačítka s „fajfkou“ nebo zrušíme stisknutím tlačítka s křížkem.

Tento program najdete na doprovodném CD v adresáři KAP07\01.



Obr. 7.2 Vizuální návrh prohlížeče databáze po aktivaci tabulky

7.2 Co, proč, jak

Náš první program tedy funguje, ale zbývá nám ještě mnoho vysvětlování. Začneme nejprve obecnými pojmy a pak přejdeme k jejich implementaci v C++ Builderu.

Databáze a podobná slova

Tabulka a databáze

Při práci s databázemi budeme neustále narážet na pojem *tabulka*. Tabulka je množina dat rozčleněných do jednotlivých *záznamů* (hovoří se také o *větách* nebo *řádcích*). Každý záznam představuje skupinu souvisejících údajů. V předchozím příkladu to např. byly údaje o jednom státu — název, hlavní město, světadíl, plocha a počet obyvatel. Jednotlivé položky v záznamu se označují jako *pole* nebo *sloupce*. Příkladem sloupce v tabulce zemí je počet obyvatel (Population).

Kdybychom záznam přirovnali ke struktuře (struct) z jazyka C, pak sloupce jsou jednotlivé složky této struktury a tabulka je množina instancí těchto struktur.

V případě tzv. stolních databází, jako je dBase, FoxPro nebo Paradox, je každá tabulka uložena v jednom nebo v několika samostatných souborech. V případě databázových serverů může být větší množství tabulek uloženo v jednom souboru — záleží na typu serveru a případně na dalších podrobnostech.

Pod pojmem databáze[42] budeme rozumět skupinu souvisejících tabulek. (Může jít např. o tabulku zákazníků, tabulku objednávek a tabulku faktur.) V případě stolních databází budou nejspíš ve společném adresáři, s případě databázových serverů závisí organizace na daném serveru.

Klíč a index

Databázovou tabulku si tedy můžeme představovat jako skutečnou tabulku, ve které jsou do řádků zapsány jednotlivé záznamy a jednotlivá pole jednotlivé složky záznamů tvoří sloupce této „skutečné“ tabulky. Budeme-li chtít záznamy v tabulce seřadit, musíme určit *klíč*. To je sloupec, podle jehož hodnot záznamy seřadíme. Klíčů může být i více; pak musíme stanovit pořadí významnosti. (Například: záznamy řadíme nejprve podle příjmení, a pokud se u dvou záznamů příjmení shodují, podle jména.)[43]

Na základě zadaných klíčů se mohou vytvořit tzv. *indexy*. Indexy si můžeme představovat jako ukazatele do databáze, které umožňují snadno najít jednotlivé záznamy. V některých databázích se indexy ukládají do samostatných souborů.

Dotaz

Dotaz je nejčastější operací, kterou s databázemi děláme. Dotaz je žádost o poskytnutí záznamů nebo některých polí ze záznamů, které odpovídají jistým požadavkům. Přitom dotaz může čerpat data z jedné nebo několika tabulek. Příkladem dotazu může být žádost o zjištění hlavního města země s největším počtem obyvatel, o zjištění všech zemí v jižní Americe nebo o zjištění všech zemí, jejichž jméno začíná písmenem G.

Výsledkem dotazu je množina dat, která se chová podobně jako tabulka.

Dotazy se v relačních databázích zpravidla vyjadřují v jazyce SQL, o kterém budeme hovořit v příští kapitole.

Transakce

Databázové programy jsou programy jako každé jiné — mohou být neočekávaně přerušeny při výpadku proudu, zhroucení operačního systému, přerušení síťového spojení a z mnoha dalších příčin. Ovšem pokud dojde k takovému incidentu uprostřed operace s daty, mohou být následky velice závažné.

Představte si, že se např. zpracovává převod peněz z jednoho účtu na druhý. To znamená, že se převáděná částka musí z jednoho účtu odečíst a k druhému přičíst. Pokud by ale program po peníze z jednoho účtu odečetl a pak se z jakéhokoli důvodu zhroutil, znamenalo by to, že se převáděná částka prostě ztratila.

[42] Pojem *databáze* je ve skutečnosti značně mlhavý. Občas — zejména ve starší literatuře — se používá ve smyslu *tabulka*. Nicméně my se budeme držet zde uvedeného vymezení.

[43] Obecněji se za klíč někdy považuje hodnota nějaké funkce vypočítaná na základě hodnot vybraných polí v záznamu.

Aby se podobným nepříjemnostem zabránilo, zavádí se v souvislosti s databázemi pojem *transakce*. To je skupina operací, která se buď musí provést jako celek, nebo se neprovede vůbec — nic mezi neexistuje. Dokončení, uzavření transakce se v programátorské hantýrce zpravidla označuje jako „komitování“ (z anglického commit), zrušení neúspěšné transakce jako „rolbek“ (z anglického roll back).

Zámek

Databázové programy pracují s daty uloženými v externích souborech. Přitom s daty v jednom souboru může pracovat i několik programů zároveň, a proto zde mohou vzniknout podobné problémy jako v multithreadových programech. Zkuste si např. představit, že v databázi je záznam o penězích na účtu X, přičemž jeden program má za úkol z tohoto účtu vybrat 1000 Kč, zatímco druhý má vložit 1000 Kč. Pokud se do toho pustí oba zároveň, může se stát, že první program si otevře příslušnou tabulku a přečte si, že na účtu X je 10000 Kč. Od této částky odečte 1000, takže dostane 9000 Kč. Než to ale stačí zapsat, přečte si druhý program, že na účtu je 10000 Kč, přičte 1000, takže dostane 11000 Kč. Mezi tím první program zapíše svůj výsledek, 9000 Kč, a nakonec zapíše svůj výsledek i druhý program (11000). Skutečný zůstatek ale měl být 10000 Kč.

Je tedy jasné, že přístup do tabulky (nebo alespoň k jednotlivým záznamům) je třeba synchronizovat. K tomu slouží tzv. zámky. Databázové servery umožňují zpravidla zamykat jednotlivé záznamy, u stolních databází lze obvykle uzamknout jen celou tabulku.

Kurzor

Pod tímto pojmem budeme rozumět jakýsi ukazatel na aktuální záznam v nějaké množině dat — např. v tabulce nebo ve výsledku dotazu.

Alias

Alias je symbolické označení, pod kterým se skrývají informace potřebné pro připojení k databázi. V případě stolních databází to je cesta do odpovídajícího adresáře.

Alias není obecný databázový pojem; používá se v borlandském databázovém stroji[44] (Borland Database Engine, BDE). Můžeme si ho vytvořit v době návrhu programu pomocí BDE Administratoru nebo Database Desktopu; později si ukážeme jak. Ve verzi Borland C++ Builder Enterprise můžeme také použít SQL Explorer. Za běhu programu můžeme vytvořit alias voláním metody TSession::AddStandardAlias() nebo TSession::AddAlias(). První vytvoří „standardní“ alias, tj. alias pro stolní databáze, druhá se používá pro připojení k databázovému serveru.

Alias umožňuje vytvářet programy nezávislé na skutečném umístění databáze — přemístíme-li databázi a vytvoříme-li pro ni alias se stejným jménem, není třeba program měnit.

Databázová architektura C++ Builderu

V předchozím příkladu jste si mohli všimnout, že C++ Builderu jsou k disposici dvě základní skupiny komponent pro vytváření databázových aplikací. První skupinu tvoří komponenty, které pracují s datovými zdroji (Table, Query, StoredProc a další). Najdeme je na paletě Data Access.

Druhou skupinu tvoří komponenty, které data zobrazují (DBGrid, DBEdit a další; anglicky se označují jako data-aware controls). Najdeme je na paletě Data Controls. Propojení mezi nimi má na starosti komponenta DataSource, kterou najdeme na paletě DataAccess.

Pod těmito komponentami jsou ovšem ještě další, nižší vrstvy, které sice přímo nepoužíváme, ale o kterých je třeba vědět. Princip práce s databázemi v C++ Builderu ukazuje obrázek 7.2.

Bezprostřední kontakt s databází tedy obstarává knihovna BDE. Ta představuje jednotné nízkoúrovňové rozhraní pro přístup k velké většině databází. Vedle nástrojů pro práci s lokálními databázemi umožňuje prostřednictvím ODBC ovladačů připojení k databázím, které podporují OBDC, a pomocí SQL spojů (SQL link) také připojení ke vzdáleným databázovým serverům. Přitom využívá síťových protokolů jako TCP/IP, SPX/IPX nebo NetBUI.

[44] Oficiální překlad používaný českým a slovenským zastoupením firmy Inprise je „borlandské databázové jádro“.

Práce s touto knihovnou „zapouzdřena“ do nevizuálních komponent pro přístup k datům, jako je Table, Query nebo StoredProc, a proces připojení se skrývá ve vytvoření a zadání aliasu. Skupinu připojení k databázi lze jako celek zapouzdřit do komponenty Session.

O přenos dat k vizuálním komponentám se vždy stará komponenta DataSource.

Tyto nevizuální komponenty se v rozsáhlejších aplikacích nevkládají do formulářů, ale do samostatných „datových modulů“.

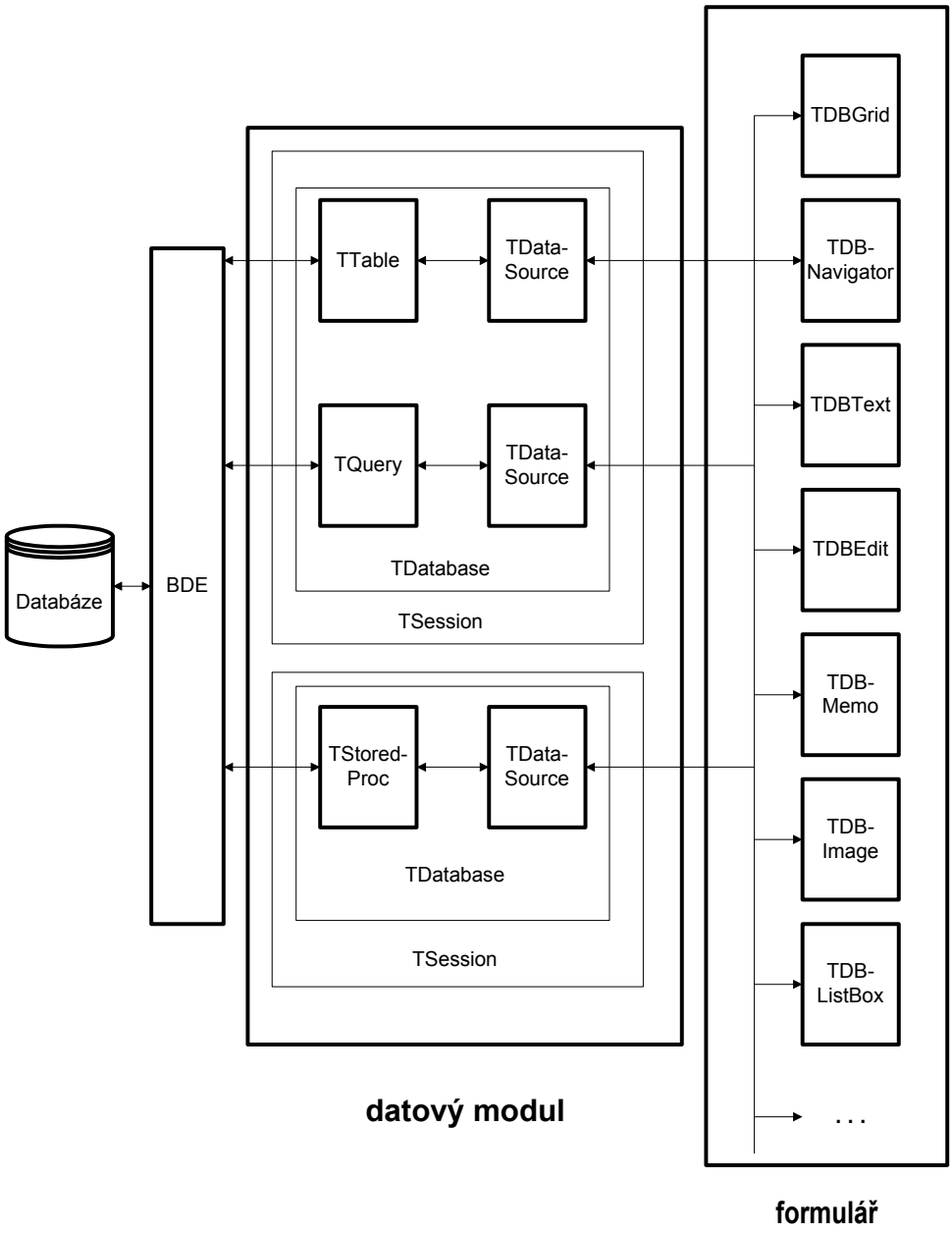
Poslední vrstvu tvoří vizuální komponenty, které data zobrazují ve formulářích.

Vizuální databázové komponenty (paleta Data Controls)

Vedle mřížky (DBGrid) a navigátoru (DBNavigator) sem patří řada komponent pro zobrazování a editování obsahu jednotlivých polí databázových záznamů. Jde o komponenty DBText, DBEdit, DBImage, DBCheckBox, DBMemo, DBRadioGroup a další, které nabízejí různé způsoby zobrazení dat. Jde o analogie komponent z palety Standard. Navíc u většiny z nich najdeme vlastnost DataSource, která určuje komponentu DataSource, od níž budou zobrazovaná data přebírat, a vlastnost DataField, která specifikuje pole (sloupec), jehož data budou zobrazovat. Přímý přístup k poli, které tato komponenta zobrazuje, lze získat pomocí vlastnosti Field, která je určena pouze ke čtení.

Komponenta DBGrid

Komponenta DBGrid zobrazuje všechna pole datového zdroje, a proto nemá vlastnost DataField. Pro úpravu zobrazených dat lze využít vlastnost Columns, která umožňuje měnit valstnosti jednotlivých sloupců, jejich nadpisy atd. Podrobnosti uvidíme na příkladech.



Obr. 7.3 Databázová architektura C++ Builderu

Komponenta DBNavigator

Také tato komponenta se vymyká schématu, o kterém jsme hovořili výše. Slouží k procházení databáze. O významu jednotlivých tlačítek jsme hovořili v komentáři k příkladu, takže zde jen doplníme, že vlastnost VisibleButtons umožňuje určit, která z tlačítek budou viditelná, a tak omezit možnosti uživatele programu. Vlastnost ConfirmDelete určuje, zda si před vymazáním záznamu z databáze program vyžádá potvrzení.

Komponenty pro přístup k datům

Do této skupiny patří především komponenty Table, Query a StoredProc, které představují zdroje dat. Jejich společným předkem je třída TDBDataSet (množina dat). Mezi vlastnostmi najdeme Database, která určuje jméno databázové komponenty, a DatabaseName, která určuje jméno databáze (adresář nebo alias), se kterou pracuje. Vlastnost CanModify určuje, zda lze získaná data měnit. Vlastnost Filter obsahuje filtr určující záznamy, které tato komponenta poskytne aplikaci.

Vlastnost Active určuje, zda je databáze otevřena, a vlastnosti Bof a Eof říkají, zda je databázový kurzor nastaven na první, resp. poslední záznam v databázi. Další vlastnosti obsahují počet sloupců v tabulce, jejich definice atd. (Tyto vlastnosti jsou k disposici jen za běhu programu.)

Mezi metodami stojí za zmínku FindFirst(), FindLast(), FindNext() a FindPrior(), které umožňují umístit kurzor na první, poslední, následující a předcházející záznam, Delete(), která vymaže aktuální záznam a umístí kurzor na následující, Append(), která k databázi připojí nový prázdný záznam, a mnohé další; jejich výčtem bychom nahrazovali nápovědu.

Mezi událostmi najdeme BeforeClose, BeforeOpe, AfterClose a AfterOpen, které nastávají před otevřením (uzavřením) datové množiny, resp. po něm, OnCalsFields, která řídí výpočet hodnot počítaných polí a mnohé další.

Vedle toho najdeme na paletě Data Access také komponentu Database. Ta slouží k zapouzdření jednotlivého připojení aplikace k databázi. Umožňuje mj. vytvořit trvalé připojení k databázi, řídit transakce a pracovat s aliasy. Pokud ve svém programu explicitně nepoužijeme komponentu Database, C++ Builder si ji vytvoří sám.

Komponenta Session představuje zapouzdření skupiny připojení k databázi. Umožňuje pracovat s aliasy a s různými databázemi atd. Pokud ve svém programu explicitně nepoužijeme komponentu Session, C++ Builder si ji vytvoří sám.

7.3 Vylepšujeme první program

Nyní se vrátíme k našemu prvnímu programu a upravíme ho.

Zákaz změn v tabulce

Náš první databázový program umožňuje měnit data v tabulce; protože ale nechceme, aby uživatel tato data měnil, zrušíme odpovídající tlačítka v navigátoru. Vybereme tuto komponentu, v inspektoru objektů vyhledáme vlastnost VisibleButtons a kliknutím na znaménko + před jejím názvem ji rozvineme. U prvních 4 dílčích vlastností ponecháme hodnotu true, u ostatních nastavíme false. Výsledkem bude, že navigátor bude obsahovat pouze tlačítka pro procházení, nikoli tlačítka pro změny (obr. 7.5).

Mohli bychom také přiřadit vlastnosti ReadOnly tabulky hodnotu false. Tato možnost je univerzálnější, neboť zakazuje změny tabulky i případným dalším komponentám a částem programu, avšak zbytečná tlačítka musíme z navigátoru stejně odstranit (a v našem případě stejně jinou možnost měnit data nemáme).

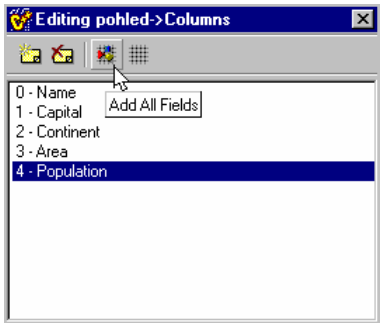
Vzhled mřížky

Databázová mřížka je velice důmyslná komponenta. Jak vidíte, nejen že umí zobrazovat data z tabulky už v době návrhu, ale sama si zjistí, kolik je v tabulce sloupců a jaké mají názvy a podle toho vytvoří i nadpisy sloupců. To je sice skvělé, ale ne vždy — v našem případě jsou nadpisy anglicky, a to se nám nelíbí. Vedle toho by asi bylo rozumné upravit šířky sloupců, neboť zabírají příliš mnoho místa.

K tomu využijeme vlastnosti Columns. Kliknutím na tlačítko se třemi tečkami vedle hodnoty této vlastnosti vyvoláme editor sloupců a stisknutím tlačítka Add All Fields do něj přidáme všechny sloupce. (Jen sloupce, které zde uvedeme, budou v mřížce zobrazeny.)

Pokud se nám nelíbí pořadí sloupců, můžeme je zde změnit prostým přetažením myší. Tím se změní jejich pořadí v mřížce. Jestliže kliknutím vybereme některý ze sloupců, můžeme jej z mřížky odstranit pomocí tlačítka Delete Selected (druhé zleva).

Inspektor objektů automaticky zobrazí vlastnosti vybraného sloupce. Zde můžeme měnit zarovnání textu ve sloupci, barvu textu, písmo, šířku sloupce (vlastnost Width), jméno sloupce tabulky, ze kterého čerpá data atd. My si najdeme vlastnost Title, která popisuje vlastnosti nadpisu sloupce, a rozvineme ji. Změnou dílčí vlastnosti Caption změníme text nadpisu. (Dále zde můžeme měnit zarovnání, písmo atd.)



Obr. 7.4 Editor sloupců v databázové mřížce

Zde tedy nahradíme implicitní nadpisy jejich českými překlady a upravíme šířku sloupců, aby se celá mřížka vešla do okna a ještě zbylo místo. Výsledek je vidět na obr. 7.5.

Filtr

Databázové programy obvykle slouží k vyhledávání specifických dat v množství obecných informací uložených v jedné nebo v několika tabulkách. Než se pustíme do vytváření dotazů, ukážeme si možnost založenou na vlastnosti Filter, kterou má nejen tabulka (komponenta Table), ale i další datové zdroje. Upravíme program tak, aby uměl vyhledávat země podle zadaného počtu obyvatel a podle názvu.

Filtr je znakový řetězec (typu AnsiString), který vyjadřuje podmínky pro některé sloupce tabulky (nebo obecně datového zdroje). Skládá se z relací obsahujících jméno sloupce, např. Area > 100000 nebo Name = 'Jamaica'. (Znakové řetězce musíme uzavřít mezi apostrofy, nikoli mezi uvozovky!) Je znakových řetězcích můžeme používat i žolík, tj. znak * zastupující skupinu libovolných znaků. Např. 'C*' znamená libovolný řetězec začínající písmenem C. Pro vyjádření relací se používají operátory <, >, <=, >=, =, <>. (Poslední dvě možnosti znamenají „rovná se“ a „nerovná se“.)

Filtr může obsahovat i několik podmínek spojených spojkami and (konjunkce, obě podmínky platí zároveň), případně or (disjunkce, platí alespoň jedna z podmínek). Složitější výrazy lze uzavřít do závorek. Negaci podmínky vyjádříme pomocí operátoru not.

Pokud jméno pole obsahuje mezery, uzavřeme je do hranatých závorek [a].

Vlastnost FilterOptions umožňuje mj. nastavit, zda se má při porovnávání brát v úvahu velikost písmen.

Aby se filtr při výběru dat z datového zdroje uplatnil, musí mít vlastnost Filtered hodnotu true; její implicitní hodnota je však false.

Po tomto výkladu se můžeme pustit do úpravy programu.

Do vizuálního návrhu formuláře vložíme komponentu GroupBox, kterou nadepíšeme Filtr, a do ní dvě editační pole, kterým přidělíme jména ed_plocha a ed_nazev. K nim umístíme dvě komponenty Label s nápisy Plocha a Název. Tato dvě editační pole budou sloužit k zadávání filtrů pro výběr záznamů podle jména a podle plochy. Pod ně umístíme tlačítko (komponentu Button) s nápisem Najdi, které pojmenujeme tl_najdi.

Pod databázový navigátor umístíme další tlačítko, které nazveme tl_vse. Bude sloužit ke zrušení nastavených filtrů. (Viz obr. 7.5.)

Budeme předpokládat, že uživatel zadá podmínku pro vyhledávání státu podle plochy ve tvaru =123456 (chce stát, který má plochu přesně 123456 km^2) nebo ve tvaru >12345 (požaduje státy, které mají plochu větší než 12345 km^2). Podobně při zadávání jména státu musí zadat např. ='V*' (požaduje všechny státy, jejichž jméno začíná písmenem V).

V handleru, který bude reagovat na stisknutí tlačítka Najdi, doplníme text podmínky získaný z filtru jménem pole. Pokud uživatel zadá obě podmínky, spojíme je spojkou and — požadujeme, aby platily zároveň.

Náš první pokus o handler THlavniOkno::tl_najdiClick() bude vypadat takto:

```
void __fastcall THlavniOkno::tl_najdiClick(TObject *Sender)
{
  AnsiString filtr;                // V této proměnné sestavíme filtr
  if(ed_plocha->Text != "") filtr = AnsiString("Area")+ed_plocha->Text;
  if(ed_nazev->Text != "")     // Sestavujeme filtr z textů v editačních polích
  {
   if(ed_plocha->Text != "") filtr += " and ";
```

```
    filtr += AnsiString("Name")+ed_nazev->Text;
  }
  tabulka->Filter=filtr;      // Nastavíme filtr jako vlastnost tabulky
  tabulka->Filtered = true;    // Zapneme filtr
}
```

Z textů v editačních polích sestavíme filtr a zapneme ho

Zde nejprve zjistíme, zda uživatel zadal nějaký text do pole ed_plocha. Pokud ano, vložíme do proměnné filtr nejprve řetězec "Area" (jméno pole v tabulce) a pak text z editačního pole. Pokud obsahují obě editační pole nějaký text, připojíme k obsahu proměnné filtr řetězec " and ", a pokud uživatel zadal nějaký text do pole ed_nazev, připojíme k obsahu proměnné filtr ještě řetězec "Name" (opět jméno pole v databázové tabulce) a text z editačního pole ed_nazev.

To znamená, že pokud např. uživatel zadá do pole Název řetězec >'C*' a pole Plocha ponechá prázdné, vznikne filtr Name > 'C*'(všechny státy, které jsou v abecedě za samotným C).

Nakonec vytvořený filtr přiřadíme vlastnosti Filter komponenty tabulka a zapneme ho (vlastnosti Filtered přiřadíme hodnotu true).

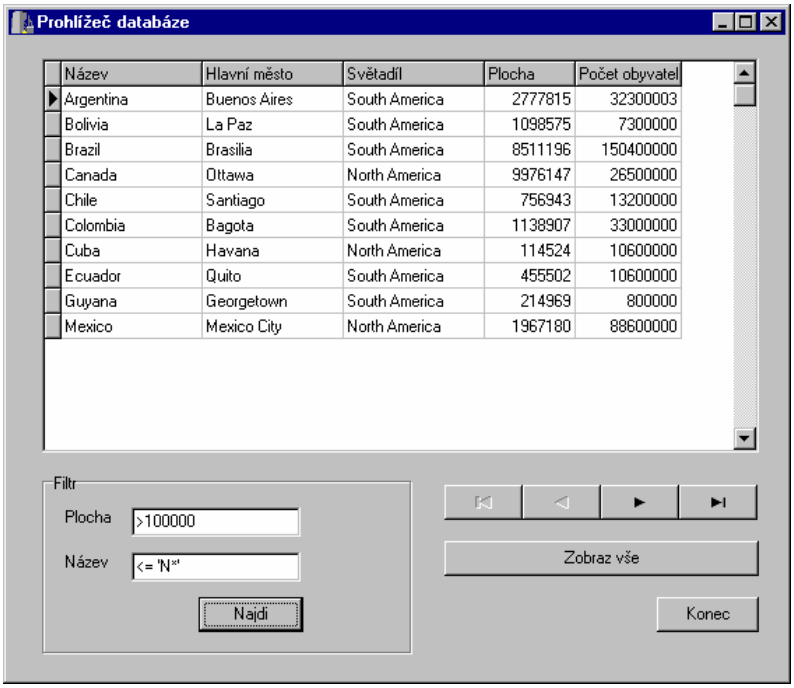
Výsledek jednoho takového dotazu ukazuje obrázek 7.4.

Po několika pokusech ovšem zjistíme problémy. První z nich je, že pokud se uživatel při zadávání filtru splete, program vypíše poněkud nesmyslné anglické hlášení. Jestliže vlastnosti Filter přiřadíme chybný řetězec, vznikne výjimka EDatabaseError. Nezbývá tedy, než tuto výjimku zachytit a ošetřit.

Nejjednodušší možnost ukazuje schématicky následující výpis.

```
void __fastcall THlavniOkno::tl_najdiClick(TObject *Sender)
{
 // ...
 // Vytvoříme filtr stejně jako předtím
  try{
   tabulka->Filter=filtr;          // Zkusíme ho nastavit
   tabulka->Filtered = true;
  }
  catch(...)                                  // a pokud se to nepovede,
  {                                                    // vypíšeme upozornění
   Application->MessageBox(filtr.c_str(),
                  "Špatně zadaný filtr", MB_OK | MB_ICONEXCLAMATION);
   ed_nazev->Text = "";            // a odstraníme chybný text
   ed_plocha->Text = "";
  }
}
```

Upravený handler, který reaguje na stisknutí tlačítka Najdi (soubor okno1.cpp)



Obr. 7.5 Státy s plochou větší než 100000 km^2, které jsou v abecedě před písmenem N

Sami se můžete pokusit o rafinovanější úpravu handleru. Je ale jasné, že filtry se hodí spíše jen pro jednoduchá vyhledávání, kdežto pro složitější zpracování bude vhodnější použít komponentu Query.

Další problém, kterého jste si při podrobnějším pohledu nejspíš také všimli, se týká třídění. Jména zemí jsou zřejmě setříděna podle anglické abecedy (písmeno „ch“ je řazeno pod „c“). K tomu se vrátíme v příští kapitole.

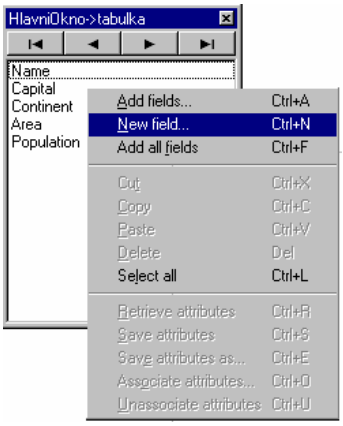
Náš program v této podobě najdete na doprovodném CD v adresáři KAP07\02.

7.4 Počítaná pole

Databázové tabulky zpravidla obsahují jen naprosto nezbytné údaje; ostatní informace se z nich v případě potřeby dopočítávají. My se pokusíme vylepšit náš program tak, aby vedle informací z tabulky ukazoval u každé země také hustotu osídlení, tj. počet obyvatel na 1 čtvereční kilometr. To znamená, že do tabulky přidáme tzv. počítané pole.

Nejprve ve vizuálním návrhu naší aplikace vybereme komponentu tabulka představující databázovou tabulku. Klikneme na ni pravým tlačítkem myši a tím vyvoláme příruční nabídku, v níž použijeme příkaz Fields Editor... (je hned první).

Objeví se prázdné okno editoru polí. Klikneme na něm pravým tlačítkem myši a tím opět vyvoláme příruční nabídku, v níž použijeme příkaz Add All Fields. Tím se do tohoto okna okopírují názvy všech polí z naší tabulky (obr. 7.6).



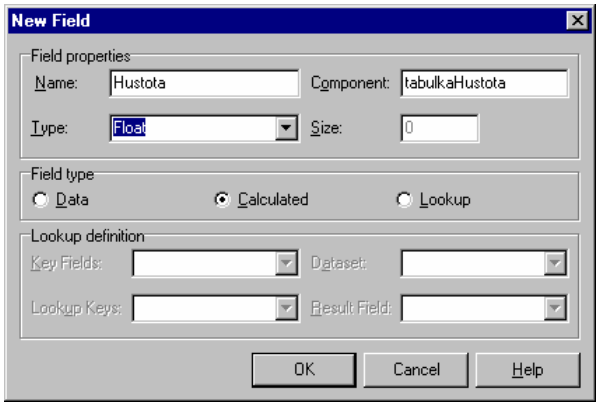
Obr. 7.6 Přidáváme nové pole do databázové tabulky

Poté vyvoláme příruční nabídku ještě jednou a tentokrát použijeme příkaz New field... (nové pole). Objeví se dialogové okno New Field (obr. 7.6). Do pole Name ve skupině Field Properties napíšeme Hustota (to bude jméno nového pole) a v seznamu u pole Type zvolíme jako typ pole Float. V poli Component se objeví text tabulkaHustota; to bude jméno komponenty, která novému poli odpovídá.[45]

Pod těmito editačními poli je skupina přepínačů nadepsaná Field Type (typ pole). V ní je implicitně vybrána možnost Calculated, tedy počítané pole. To přesně odpovídá našim požadavkům, a proto zde nic nebudeme měnit. Další možnost, Data, znamená nové datové pole; obvykle nahrazuje existující datové pole, např. při změně typu. Možnost Lookup znamená pole, které zobrazuje hodnoty vybrané podle určitých kritérií z datové množiny určené za běhu programu.

Nyní musíme přidat nový sloupec také do mřížky, která data zobrazuje. Ve vizuálním návrhu programu tedy vybereme mřížku, pomocí pravého tlačítka myši vyvoláme příruční nabídku a v ní vybereme opět položku Column Editor.... Stisknutím klávesy INS (nebo pomocí příruční nabídky) sem přidáme nový sloupce, který dostane implicitní označení TColumn.

V inspektoru objektů pak definujeme vlastnosti nového sloupce. Ve skutečnosti jediná věc, kterou musíme určit, je jméno pole v databázi, ze kterého bude čerpat data, a to je určeno vlastností FieldName. Z rozbalovacího seznamu u této vlastnosti vybereme Hustota. Poté se v mřížce objeví nový sloupec s nadpisem Hustota, ovšem bez údajů.



Obr. 7.7 Dialogové okno pro přidání nového pole do tabulky

Nyní zbývá poslední úkol — přimět databázovou tabulku, aby si vypočítala hodnoty počítaného pole. K tomu nám poslouží handler ošetřující událost OnCalcFields, která nastává, je-li třeba přepočítat hodnoty počítaných polí.

K výpočtu polí nám poslouží vlastnost Value komponent představujících pole databáze. Tato vlastnost obsahuje hodnotu pole. Uvedený handler tedy bude

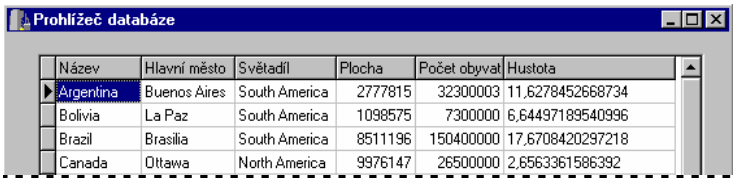
```
void __fastcall THlavniOkno::tabulkaCalcFields(TDataSet *DataSet)
{
 tabulkaHustota->Value = tabulkaPopulation->Value/(float)tabulkaArea->Value;
```

[45] Ostatní komponenty odpovídající jednotlivým sloupcům tabulky se budou jmenovat tabulkaName, tabulkaArea atd.

}

Výpočet hustoty obyvatelstva v poli Hustota (soubor okno1.cpp)

Výsledek uvidíme až za běhu; v době návrhu se počítaná pole nezobrazují. Ovšem pak zjistíme, že se výsledky zobrazují se zbytečnou přesností. Navíc pokud sloupec Hustota dostatečně rozšíříme, zjistíme, že hodnoty v něm jsou zarovnány doleva, a to je u čísel neobvyklé (obr. 7.7).

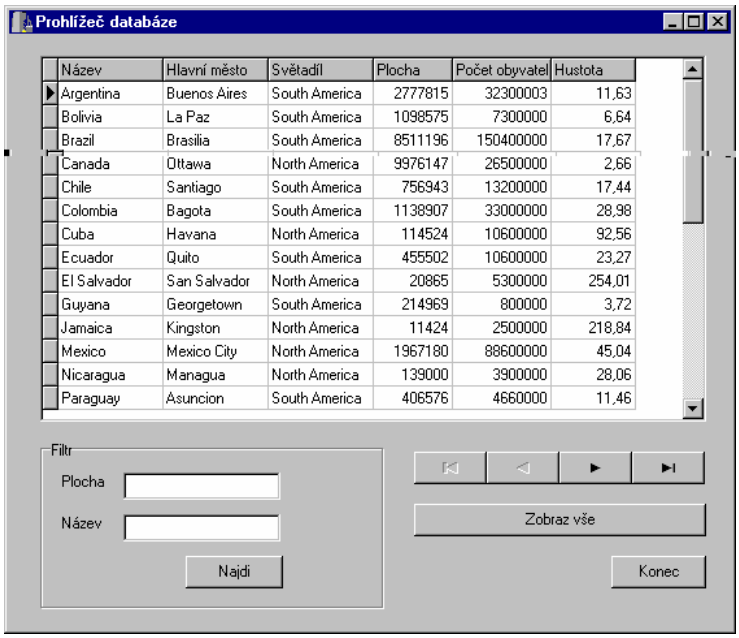


Obr. 7.8 Hustota obyvatelstva s přesností na 13 desetinných míst? To nemá smysl…

Nejprve tedy vyhledáme v seznamu inspektora objektů komponentu tabulkaHustota, která představuje odpovídající sloupec tabulky, a její vlastnosti DisplayFormat přiřadíme hodnotu ##0.00. Tím řekneme, že chceme před desetinnou čárkou nejméně jednu, nejvýše však tři číslice, za desetinnou čárkou vždy dvě číslice. (Pozor, sloupec tabulky má také vlastnost EditFormat. Ta však určuje formátování při editování, nikoli při zobrazování.)

Pak vybereme mřížku, vyvoláme editor sloupců mřížky, vybereme sloupec Hustota a v inspektoru objektů přidělíme vlastnosti Alignment hodnotu taRightJustify. Tím zarovnáme obsah sloupce doprava. Výsledek ukazuje obrázek 7.8.

Tento program najdete na doprovodném CD v adresáři KAP07\03.



Obr. 7.9 Hustota obyvatelstva, tentokrát na dvě desetinná místa a zarovnaná vpravo

Poznámka

Formát, ve kterém se data ve sloupci tabulky zobrazí, nastavujeme ve vlastnosti DisplayFormat jako řetězec složený z několika druhů znaků. Zde si řekneme jen základní informace, podrobnosti najdete v nápovědě.

znak	**význam**
#	Číslice. Na udané pozici ve výstupu se zobrazí buď číslice nebo mezera.
0	Číslice. Pokud by se na udané pozici ve výstupu měla zobrazit mezera, nahradí se nulou.
. (tečka)	Desetinná tečka nebo čárka. Skutečně použitý znak je určen globální proměnnou DecimalSeparator (a její implicitní obsah se řídí lokálním nastavením). První znak „.“ určuje polohu desetinné tečky ve výstupu.

znak	význam
, (čárka)	Oddělovač tisíců. Skutečně použitý znak je určen globální proměnnou ThousandSeparator.
E+	Semilogaritmický zápis.
"xx", 'xx'	Znaky uzavřené v apostrofech nebo uvozovkách se beze změny okopírují do výstupu.
;	Odděluje specifikaci formátu kladného a záporného čísla a nuly.

7.5 Tabulky master-detail

Často potřebujeme pracovat s daty z několika tabulek, a to tak, že v jedné (hlavní, řídící) tabulce jsou základní údaje a ve druhé (podřízené) tabulce jsou doplňující údaje. Představme si např., že máme píšeme program pro obchodní firmu, která má ve své databázi tabulku zákazníků a tabulku objednávek. Naším úkolem je vytvořit program, který umožní prohlížet si tabulku zákazníků a přitom pro každého zákazníka ihned zobrazí jeho vyřízené i nevyřízené objednávky.

Toto uspořádání se obvykle označuje *master-detail* a jeho programování v C++ Builderu není o mnoho složitější než to, co jsem dosud dělali. Ve skutečnosti nepůjde o nic víc, než že v podřízené tabulce nastavíme vlastnost MasterSource (zdroj dat, přes který se napojíme na hlavní tabulku) a MasterFields (pole, podle kterých budeme určovat data vybíraná z podřízené tabulky).

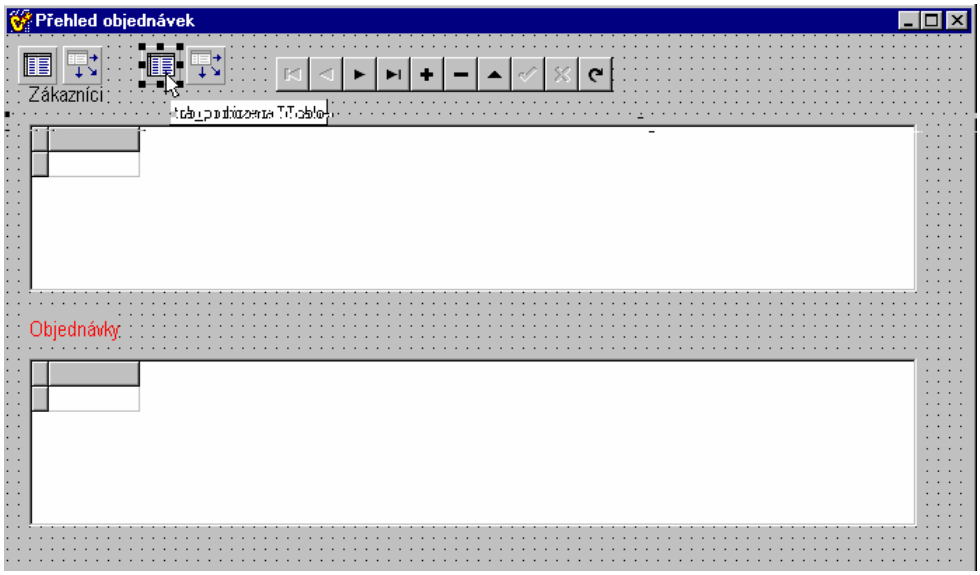
Potřebné tabulky jsou opět součástí instalace C++ Builderu pod aliasem BCDEMOS a jmenují se customer.db (zákazník) a order.db (objednávka).

Začneme nový projekt, který uložíme pod názvem DB4. Modul hlavního okna pojmenujeme jako obvykle okno1. Do vizuálního návrhu okna vložíme komponentu Table, kterou pojmenujeme tab_hlavni. Její vlastnosti DatabaseName přiřadíme hodnotu BCDEMOS a vlastnosti TableName hodnotu customer.db. Pak vložíme do vizuálního návrhu komponentu DataSource, pojmenujeme ji ds_hlavni a připojíme ji k tabulce tab_hlavni. Poté sem vložíme komponentu DBGrid, kterou pojmenujeme gr_hlavni a kterou napojíme (vlastnost DataSource) na komponentu ds_hlavni. Nakonec sem vložíme komponentu DBNavigaror, kterou pojmenujeme nav_hlavni a kterou také napojíme na komponentu s_hlavni. (Zatím jsme vytvořili podobný program jako v první části této kapitoly — prohlížeč jedné tabulky. Vlastnosti Active tabulky jsme ovšem ponechali hodnotu false.)

Nyní do vizuálního návrhu programu přidáme další komponentu Table, kterou pojmenujeme tab_podrizena. Její vlastnosti DatabaseName přiřadíme stejně jako u první tabulky hodnotu BCDEMOS a vlastnosti TableName hodnotu order.db. Pak vložíme do vizuálního návrhu komponentu DataSource, pojmenujeme ji ds_podrizena a připojíme ji k tabulce tab_podrizena. Poté sem vložíme komponentu DBGrid, kterou pojmenujeme gr_podrizena a kterou napojíme (vlastnost DataSource) na komponentu ds_podrizena.

Nakonec můžeme přidat několik vysvětlujících nápisů (komponenty Label). Vizuální návrh tohoto programu ukazuje obr. 7.9.

Zatím máme v programu dvě nezávislé tabulky. Nyní mezi nimi definujeme vztah master-detail. Ve vizuálním návrhu vybereme tabulku tab_podrizena, v insprktoru objektů vyhledáme vlastnost MasterSource a přiřadíme jí hodnotu ds_hlavni (ostatně nic jiného nám rozbalovací seznam u této vlastnosti nenabídne). Tím specifikujeme, že tab_podrizena je podřízená tabulka a výběr dat z ní bude řízen daty z komponenty DataSource jménem ds_hlavni.



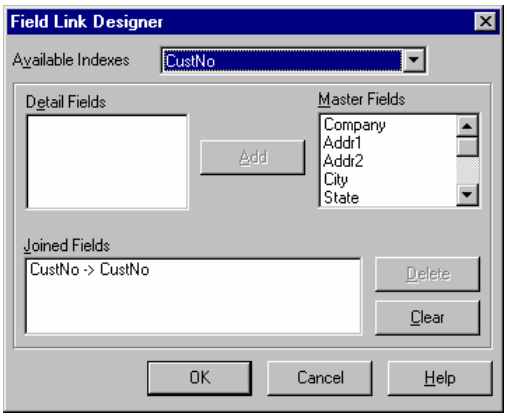
Obr. 7.10 Vizuální návrh pro tabulky ve vztahu master-detail

Nyní musíme ještě říci, které pole bude výběr řídit. V našem případě použijeme v obou tabulkách pole CustNo (číslo zákazníka). Jestliže bude kurzor v hlavní tabulce ukazovat např. na zákazníka číslo 1234, vyhledají se v podřízené tabulce všechny záznamy o objednávkách zákazníka s číslem zákazníka 1234 a zobrazí se. Poznamenejme, že k propojení tabulek lze použít pouze polí, pro které jsou vytvořeny indexové soubory.

Vyhledáme tedy v inspektoru objektů vlastnost MasterFields, vybereme ji a stisknutím tlačítka se třemi tečkami vedle hodnoty této vlastnosti vyvoláme dialogové okno Field Link Designer (návrhář propojení polí). V poli Available Indexes zvolíme CustNo, pole z podřízené tabulky, které chceme použít k propojení. Pak v polích Detail Fields a Master Fields vybereme myší položku CustNo a stiskneme tlačítko Add. Tím jsme definovali propojení těchto dvou polí a toto propojení se objeví v poli Joined Fields (obr. 7.11).

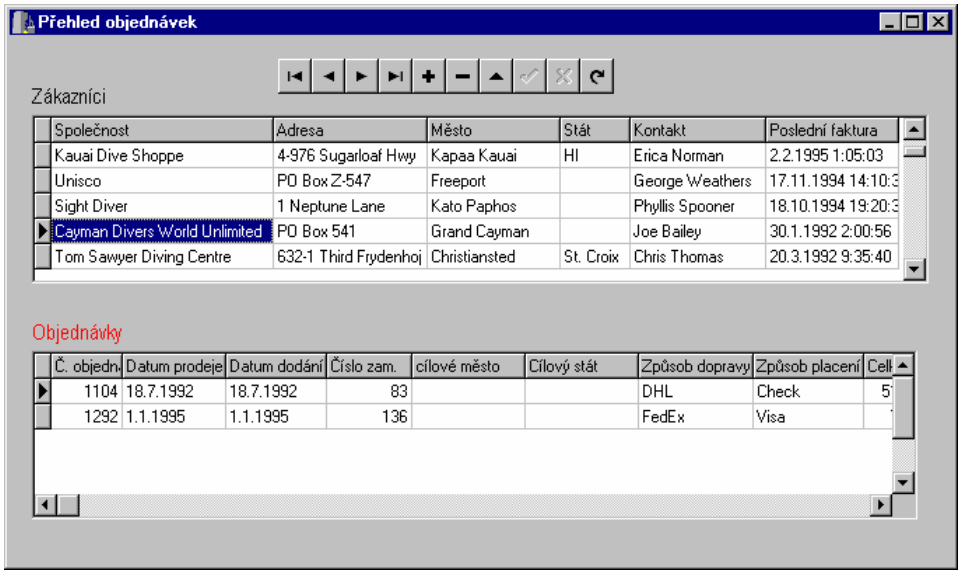
Stisknutím OK návrhář propojení uzavřeme.

Nyní se vrátíme k tabulkám a u obou přiřadíme vlastnosti Active hodnotu true. Tím jsme hotovi a můžeme program přeložit a spustit.



Obr. 7.11 Definice propojení polí v návrháři propojení

Program v této podobě (s několika kosmetickými úpravami, jako je vypuštění některých sloupců, úprava šířky sloupců, přeložené nadpisy apod.) najdete na doprovodném CD v adresáři KAP07\04. Jeho okno za běhu ukazuje obr. 7.12.



Obr. 7.12 Tabulky v uspořádání master-detail za běhu

8. Ještě jednou databáze

Začneme tím, že se podíváme na několik příkladů, ve kterých použijeme v databázích české názvy polí a ve znakových řetězcích písmena s diakritickými znaménky. Nejprve se ale seznámíme se základy jazyka SQL.

8.1 SQL

Označení SQL je zkratkou slov Structured Query Language, strukturovaný dotazovací jazyk.[46] Je to nejrozšířenější jazyk používaný pro komunikaci s relačními databázemi. V současné době platí standard SQL 2, mnohé databázové servery však nabízejí řadu rozšíření, která by se měla objevit v připravovaném standardu SQL 3. V tomto oddílu poznáme opravdu jen základní konstrukce; podrobné informace lze najít např. v [10].

V SQL se obvykle rozlišují dvě podmnožiny: příkazy pro definici dat (Data Definition Language — DDL) a příkazy pro manipulaci s daty (Data Modification Language — DML). Do první skupiny patří např. příkazy CREATE, DROP a ALTER, do druhé především SELECT, INSERT, UPDATE a DELETE.[47]

Dotaz (příkaz SELECT)

Pomocí příkazu SELECT klademe dotazy — zjišťujeme data z databází. Příkaz SELECT vrátí výsledkovou množinu, tj. množinu záznamů, které splňují podmínky dotazu. Jeho strukturu lze schématicky zapsat takto:

SELECT *seznam_sloupc* FROM *seznam_tabulek*
WHERE *podmínka*
ORDER BY *klíč*
GROUP BY *klíč*
HAVING *podmínka*

Podívejme se nyní na jeho význam. Za klíčovým slovem SELECT následuje seznam databázových sloupců, které nás zajímají, a za slovem FROM pak seznam tabulek, z nichž je chceme vybírat. Chceme-li všechny sloupce, můžeme jejich výčet nahradit žolíkem *.

Za klíčovým slovem WHERE následuje podmínka; do výsledkové množiny se dostanou jen záznamy, které jim budou vyhovovat.

Za klíčovými slovy ORDER BY následuje klíč, podle kterého se budou záznamy třídit, a za GROUP BY klíč pro uspořádání do skupin za účelem výpočtu různých hodnot z dat v tabulce.

Část HAVING obsahuje podmínku, která určuje skupiny vytvořené částí GROUP BY, které se nají do výsledku zahrnout.

Části uvedené klíčovými slovy WHERE, ORDER, GROUP a HAVING lze vynechat.

V podmínkách můžeme uvádět výrazy zkonstruované pomocí operátorů <, <=, >=, >, <> a =. Poslední dva znamenají *nerovná se* a *rovná se.* Složitější výrazy lze uzavírat do závorek a spojovat logickými operátory OR (nebo, disjunkce) a AND (a zároveň, konjunkce) nebo negovat operátorem NOT.

Další operátory a funkce, které zde můžeme použít, jsou např.:

IS NULL	vrátí True (pravdivou hodnotu), jestliže hodnota v daném sloupci není definováno (uživatel tam nic neuložil),
IS NOT NULL	vrátí True (pravdivou hodnotu), jestliže je hodnota v aktuálním záznamu v daném sloupci definována,
BETWEEN	test, zda hodnota leží v daném rozmezí, např. cust_no BETWEEN 100 AND 200,

[46] Vedle české výslovnosti *es-kvé-el* nebo anglické *eskjuel* se setkáme i s výslovností *síkvel,* která pochází z označení SEQUEL (Structured English Query Language), což byl předchůdce SQL, navržený a implementovaný u IBM v první polovině 70. let.

[47] Jazyk SQL nerozeznává velká a malá písmena. Proto budeme pro usnadnění čitelnosti psát jeho klíčová slova většinou velkými písmeny.

||	spojování řetězců, např. lastname || ',' || firstname (řetězcové konstanty zapisujeme do uvozovek),
IN *seznam*	vrátí True, leží-li levý operand v daném *seznamu*,
UPPER()	převede znaky v daném řetězci na velká písmena, např. UPPER(lastname),
LIKE	vrací řetězce podobné zadanému, např. LIKE 'A%T' znamená všechny řetězce začínající A a končící T; % zde funguje jako žolík,
AVG()	vypočte průměr, např. SELECT AVG(salary) FROM employee,
MAX()	vypočte maximum, tj. největší z hodnot,
MIN()	vypočte minimum, tj. nejmenší z hodnot,
COUNT	vrátí počet hodnot ve sloupci,
DISTINCT	stojí bezprostředně za slovem SELECT a přikazuje vybírat jen různé hodnoty,
ASC, DESC	přikazuje vzestupné, resp. sestupné třídění.

Jako žolíky můžeme ve znakových řetězcích používat znak _ (podtržení), který zastupuje právě jeden libovolný znak, a znak %, který zastupuje libovolný počet libovolných znaků.

Pokud příkaz SELECT pracuje s jedinou tabulkou, stačí uvádět samotná jména sloupců. Pokud by mohlo dojít k nejednoznačnostem (různé tabulky mohou obsahovat sloupce se stejnými jmény), musíme jméno sloupce kvalifikovat jménem tabulky pomocí operátoru tečka, např. customer.cust_no**.**

Nejjednodušším příkladem tedy bude např. příkaz

```
SELECT * FROM country
```

Zde požadujeme všechna pole a neuvádíme žádné podmínky ani požadavky na třídění. Výsledkem tedy budou všechny záznamy (se všemi sloupci) z tabulky country. Příkaz

```
SELECT DISTINCT city FROM customer
```

zjistí všechna města, ve kterých máme zákazníky, a žádné město nebude opakovat. Příkaz

```
SELECT *
FROM customers
WHERE cust_no = 100 OR cust_no = 200
```

vypíše z tabulky customers zákazníky s číslem cust_no rovným 100 nebo 200. Příkaz

```
SELECT AVG(salary) from employee
```

zjistí průměrný plat zaměstnanců z tabulky employee. Příkaz

```
SELECT * FROM employee ORDER BY lastname
```

nám vrátí tabulku zaměstnanců seřazenou podle příjmení (sloupec lastname).

V podmínce můžeme použít také výsledek dalšího příkazu SELECT. Např. příkazem

```
SELECT firstname, lastname FROM employee
WHERE salary = (SELECT max(salary) FROM employee)
```

zjistí nejlépe placené zaměstnance. Příkaz

```
SELECT max(salary), avg(salary), department
FROM employee
GROUP BY department
```

zjistí nejvyšší a průměrný plat (salary) v jednotlivých odděleních (department). Poznamenejme, že v části GROUP BY se musí objevit všechny sloupce uvedené za SELECT, která nejsou výsledkem výpočtu.

Vkládání záznamů (příkaz INSERT)

Tento příkaz slouží k přidávání nových záznamů do tabulky. Ukážeme si jednoduchý příklad:

```
INSERT INTO employee (empno, lastmane, firstname, salary)
VALUES (0, "Novák", "František", 10000).
```

Do tabulky lze také vložit výsledek dotazu, tedy příkazu SELECT. Příkaz INSERT pak má tvar

```
INSERT INTO tabulka p   kaz_select
```

Aktualizace (příkaz UPDATE)

Příkaz UPDATE slouží k hromadné aktualizaci tabulky. K výběru záznamů, které chceme upravit, lze použít podmínku WHERE podobně jako v příkazu SELECT. Příkaz pero změnu hodnoty v poli tabulky uvádí klíčové slovo SET. Např. příkazem

```
UPDATE employee
SET salary = salary * 1.05
WHERE empno < 1000
```

zvýšíme všem zaměstnancům s číslem menším než 1000 plat o 5%.

Mazání záznamů (příkaz DELETE)

Tento příkaz odstraní z tabulky požadovaný záznam. Jeho tvar si opět ukážeme na příkladu. Příkaz

```
DELETE FROM employee WHERE empno = 221
```

odstraní z tabulky employee zaměstnance s číslem 221.

Vytváření a rušení tabulky (příkazy CREATE TABLE, DROP TABLE)

Chceme-li vytvořit novou tabulku, použijeme příkaz CREATE TABLE, který má tvar

```
CREATE TABLE ( popisy_sloupc );
```

Popisy jednotlivých sloupců obsahují jméno sloupce, typ a případně integritní omezení a specifikaci, že jde o primární klíč. Jednotlivé položky v popisu pole oddělujeme mezerami, popisy jednotlivých sloupců čárkami. Podívejme se na příklad:

```
CREATE TABLE zakaznik (
 cislo integer NOT NULL PRIMARY KEY,
 jmeno varchar(30) NOT NULL,
 prijmeni varchar(30) NOT NULL
 adresa varchar(30),
 telefon varchar(20)
);
```

NOT NULL znamená, že uživatel musí zadat hodnotu v tomto sloupci. Primární klíč lze zadat i samostatně, jako skupinu sloupců:

```
PRIMARY KEY (cislo, prijmeni)
```

Tento popis zapíšeme v příkazu CREATE TABLE za popisy polí. Příkazem

```
DROP TABLE jméno;
```

odstraníme tabulku z databáze.

Vytváření indexů (příkaz CREATE INDEX)

Vytvořením indexů se výrazně urychlí vyhledávání v tabulce. Indexy pro zvolený klíč vytvoříme příkazem

```
CRREATE INDEX jméno_indexu ON jméno_tabulky( kl )
```

Indexy odstraníme příkazem

```
DROP INDEX jméno.
```

8.2 Třídění

Na doprovodném CD v adresáři KAP08\TABULKY najdete databázové tabulky zakaznik a ucty, o kterých budeme předpokládat, že obsahují přehled o zákaznících jistého nejmenovaného finančního ústavu a o stavech jejich účtů. Ke skutečnému bankovnímu systému mají opravdu hodně daleko, ale to pro nás vůbec není podstatné. Jde o tabulky ve formátu pro Paradox 7. První z nich, tabulka zakaznik, obsahuje údaje o klientech, tj. sloupce Jméno, Příjmení, Město, Datum narození, Cis_zak a další. (Názvy sloupců mohou

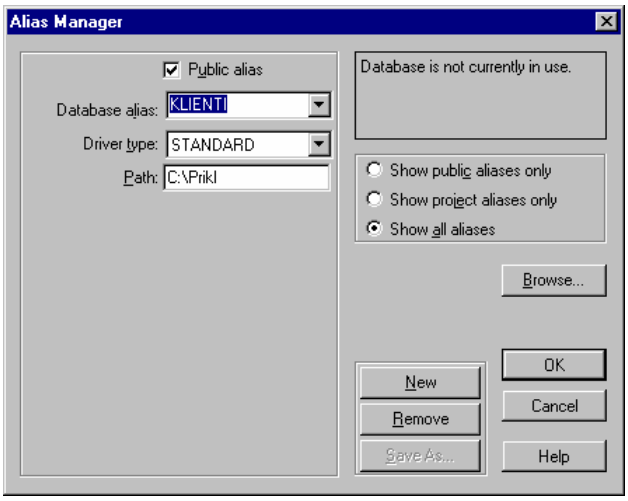
obsahovat i znaky české abecedy a dokonce i mezery.) Ve druhé tabulce, ucty, jsou informace o účtech. Její nejdůležitější sloupce se jmenují Číslo účtu, Číslo zákazníka a Stav. Přitom sloupec Číslo zákazníka z této tabulky odpovídá sloupci Cis_zak z tabulky zakaznik.

Všimněte se, že každá tabulka se vlastně skládá ze 3 souborů. Tuto soubory překopírujte do zvoleného adresáře; zde budeme pro určitost předpokládat, že jde o podadresář C:\PRIKL. (Tento předpoklad se ale uplatní jen na obr. 8.1.)

Alias

Nyní si pro tuto databázi vytvoříme alias. Z IDE C++ Builderu spustíme příkazem Tools | Database Desktop pomocný program Database Desktop. Z nabídky tohoto programu pak vyvoláme příkazem Tools | Alias Manager okno Alias Manager (obr. 8.1).

Nyní stiskneme tlačítko New. Pak zaškrtneme pole Public alias, neboť chceme veřejně přístupný alias, a do pole Database alias napíšeme jméno nově vytvářeného aliasu — např. KLIENTI. V poli Driver type zvolíme typ ovladače, který se bude pro tento alias používat. Přijmeme nabídnutou hodnotu STANDARD, neboť ta znamená standardní ovladače pro desktopové databáze Paradox a dBase. V poli Path pak zadáme cestu do adresáře. Přitom si můžeme pomoci tlačítkem Browse..., které nám zobrazí seznam aliasů a adresářů.[48] Pak stiskneme OK. Prostředí se nás ještě zeptá, zda si opravdu přejeme takto vytvořený alias uložit. Pokud odpovíme, že ano, uloží ho do souboru IDAPI32.CFG, který najdeme v adresáři ...Borland\BDE.



Obr. 8.1 Vytváříme alias

Dotaz

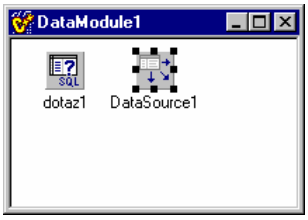
Nyní napíšeme program, který bude vypisovat jména zákazníků seřazená podle různých kritérií. K tomu se nehodí komponenta Table, neboť ta automaticky třídí data podle primárního klíče. Použijeme tedy komponentu Query a jazyk SQL.

Datový modul

Vytvoříme nový projekt, který uložíme pod jménem dotaz1; jeho hlavní okno pojmenujeme jako vždy HlavniOkno a příslušný modul okno1.

[48] C++ Builder 4 Enterprise dále standardně nabízí ovladače pro Access, DB2, FoxPro, Informix, InterBase, MS SQL, Oracle, Sybase a pro databáze přístupné prostřednictvím ODBC. Pro databázové servery je okno Alias Manager podstatně složitější, neboť obsahuje pole pro jméno serveru, uživatelské jméno, pod kterým se budeme přihlašovat, zadání různých režimů atd. Jejich vyplnění předpokládá znalost práce se serverem, ke kterému se prostřednictvím daného aliasu budeme připojovat.

Abychom od sebe oddělili komponenty, které pracují s databází, od komponent, které se starají o uživatelské rozhraní, vytvoříme si datový modul. To je třída, která funguje jako kontejner pro nevizuální databázové a jiné komponenty. Příkazem File | New... vyvoláme zásobník objektů a v něm zvolíme na kartě New položku Data Module. Prostředí otevře okno Data Module, do kterého můžeme vkládat nevizuální komponenty podobně jako do okna vizuálního návrhu, viz obr. 8.2. (Toto okno ovšem nebude za běhu programu vidět.) Zároveň vytvoří zdrojový kód modulu, který se skládá z hlavičkového souboru s definicí třídy datového modulu a ze souboru s implementací. Uložíme ho příkazem File | Save All pod názvem dm1; komponentě DataModule ponecháme nabídnuté jméno DataModule1.



Obr. 8.2 Okno datového modulu v době návrhu programu

Do okna datového modulu nyní vložíme komponentu Query, kterou pojmenujeme dotaz1, a její vlastnosti DatabaseName přiřadíme nově vytvořený alias KLIENTI.

Pak do tohoto okna přidáme komponentu DataSource, ponecháme jí nabídnuté jméno DataSource1 a její vlastnosti DataSet přiřadíme hodnotu Dotaz1.

Zdrojový text datového modulu bude nyní vypadat takto:

```
class TDataModule1 : public TDataModule
{
__published:  // IDE-managed Components
    TDataSource *DataSource1;
    TQuery *dotaz1;
private:      // User declarations
public:       // User declarations
    __fastcall TDataModule1(TComponent* Owner);
};
extern PACKAGE TDataModule1 *DataModule1;
```

Deklarace třídy datového modulu (soubor dm1.h)

Soubor s implementací obsahuje pouze definici konstruktoru a deklaraci instance:

```
TDataModule1 *DataModule1;
__fastcall TDataModule1::TDataModule1(TComponent* Owner)
    : TDataModule(Owner)
{}
```

Soubor dm1.cpp obsahuje zatím pouze konstruktor, který volá konstruktor předka

Je zřejmé, že datové moduly fungují podobně jako formuláře: Od univerzálního předka, třídy TDataModule, se odvodí potomek, který bude obsahovat ukazatele na použité komponenty a případné další složky a metody. Na rozdíl od formulářů ale datové moduly nemají za běhu programu své okno.

Mřížka

Pak vložíme do hlavního formuláře komponentu DBGrid, kterou pojmenujeme dbg_mriz. Její vlastnost DataSource musí odkazovat na komponentu DataSource1; ta je však nyní v jiném modulu, proto ji prostředí nevidí a nenabídne nám ji v rozbalovacím seznamu u jména této vlastnosti. Musíme tedy nejprve vložit do souboru okno.cpp hlavičkový soubor dm1.h[49].

SQL

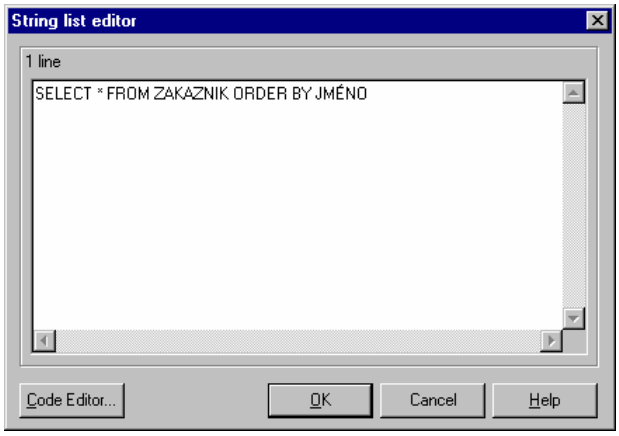
Než budeme pokračovat v psaní našeho programu, seznámíme se blíže s dotazy v SQL. Vyzkoušíme si několik různých dotazů, které budeme nejprve v době návrhu.

[49] K tomu můžeme použít příkaz File | Include Unit Hdr, který vyvolá dialogové okno se seznamem ještě nevložených hlavičkových souborů modulů našeho programu.

Dotaz v jazyce SQL je obsahem vlastnosti SQL komponenty Query. Vybereme tedy v inspektoru objektů vlastnost SQL, kliknutím na tlačítko se třemi tečkami vedle pole pro hodnotu otevřeme její editor (obr. 8.3) a zapíšeme do něj text

```
SELECT * FROM zakaznik ORDER BY jméno
```

kterým požadujeme výpis všech polí z tabulky zakaznik seřazený podle jmen (křestních). Poznamenejme, že není třeba dodržovat velká a malá písmena.



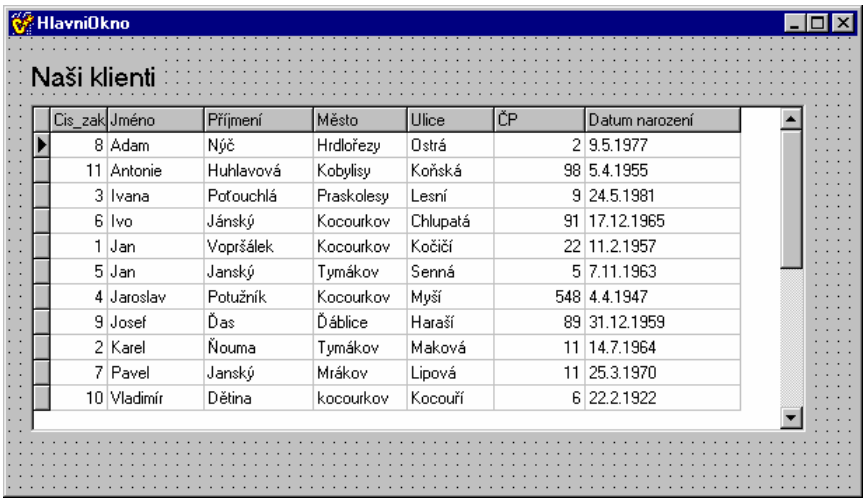
Obr. 8.3 Zadáváme dotaz v době návrhu programu

Pak toto okno uzavřeme stisknutím OK a nastavíme vlastnost Active komponenty dotaz1 na true. Dotaz se okamžitě provede a mřížka nám zobrazí jeho výsledky — a pokud máme správně nastavený databázový ovladač, budou seřazeny téměř podle pravidel českého třídění, viz obr. 8.5.

Pokusme se nyní setřídit klienty podle věku od nejmladšího po nejstaršího. Protože název pole Datum narození obsahuje mezeru a protože jde o dotaz pro lokální databázi, který bude zpracovávat BDE, musíme ho uzavřít do uvozovek a kvalifikovat jménem tabulky:

```
SELECT * FROM zakaznik
ORDER BY zakaznik."Datum narození" DESC
```

Operátor DESC přikazuje třídit v sestupném pořadí, tj. na prvním místě bude nejvyšší datum.

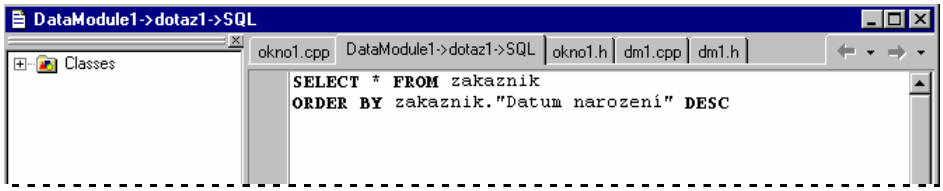


Cis_zak	Jméno	Příjmení	Město	Ulice	ČP	Datum narození
8	Adam	Nýč	Hrdlořezy	Ostrá	2	9.5.1977
11	Antonie	Huhlavová	Kobylisy	Koňská	98	5.4.1955
3	Ivana	Poťouchlá	Praskolesy	Lesní	9	24.5.1981
6	Ivo	Jánský	Kocourkov	Chlupatá	91	17.12.1965
1	Jan	Vopršálek	Kocourkov	Kočičí	22	11.2.1957
5	Jan	Janský	Tymákov	Senná	5	7.11.1963
4	Jaroslav	Potužník	Kocourkov	Myší	548	4.4.1947
9	Josef	Ďas	Ďáblice	Haraší	89	31.12.1959
2	Karel	Ňouma	Tymákov	Maková	11	14.7.1964
7	Pavel	Janský	Mrákov	Lipová	11	25.3.1970
10	Vladimír	Dětina	kocourkov	Kocouří	6	22.2.1922

Obr. 8.4 Výsledek dotazu

Poznámka

Nyní můžete chvíli experimentovat s komponentou Query a s dotazy v jazyce SQL. Abyste nemuseli neustále otevírat a zavírat okno s editorem vlastnosti SQL, stiskněte v něm tlačítko Code Editor. Text dotazu se nyní zapíše na samostatnou kartu v editoru zdrojového textu (obr. 8.5), kde je neustále dostupný. Musíte jej ale po každé změně uložit.



Obr. 8.5 Editor SQL

Po každé úpravě dotazu se automaticky nastaví vlastnost Active komponenty dotaz1 na false.

Pokud se dotaz nepodaří provést, např. proto, že obsahuje syntaktickou chybu, protože neexistuje požadovaný sloupec v tabulce ap., vznikne výjimka. Pokud se to stane v době návrhu, objeví se dialogové okno s upozorněním na druh chyby (ne vždy ovšem srozumitelným).

Kdo je nejbohatší

Jazyk SQL ovšem umožňuje pokládat i komplikovanější dotazy, ve kterých použijeme data z několika tabulek. Chceme např. zjistit, jak se jmenuje klient, který má na účtě nejvyšší částku. Ovšem údaje o účtech jsou v tabulce ucty, zatímco údaje o klientech jsou v tabulce zakaznik.

Při vytváření patřičného dotazu budeme postupovat takto:

Nejprve zjistíme, jaká je nejvyšší uložená částka, tedy nejvyšší hodnota v poli stav, dotazem

```
select max(stav) from ucty
```

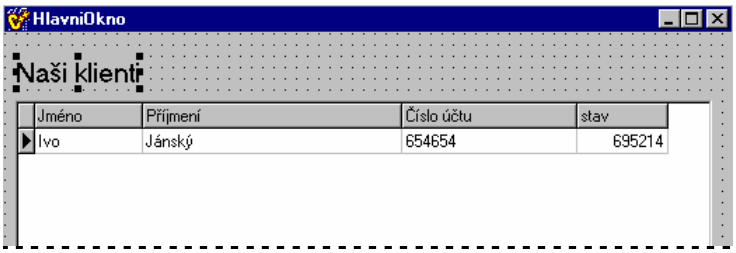
To znamená, že číslo tohoto zákazníka můžeme zjistit složeným příkazem

```
Select ucty."  slo z kazn ka" from ucty
where stav =
   (select max(stav) from ucty)
```

Nyní už můžeme zformulovat dotaz, kterým zjistíme nejbohatšího klienta:

```
select Jméno, P   jmen , ucty."  slo    tu", stav from zakaznik, ucty
where
 (zakaznik.cis_zak = (Select ucty."Číslo zákazníka" from ucty
              where stav =
                (select max(stav) from ucty)
              )
 )
 AND
 ucty."Číslo zákazníka" = zakaznik.cis_zak
```

Výsledek ukazuje obr. 8.6.



Obr. 8.6 Náš nejbohatší klient

Dotazy za běhu programu

Hlavní síla komponenty Query spočívá v tom, že umožňuje vytvářet dotazy za běhu programu podle zadání uživatele. Při změně hodnoty vlastnosti SQL ovšem musíme vždy nejprve komponentu Query „odpojit“ od databáze, tj. zavolat její metodu Close(), pak vymazat aktuální dotaz voláním metody SQL -> Clear(), pomocí metody SQL -> Add() sestavit nový dotaz a nakonec ho spustit tím, že zavoláme metodu SQL -> Open().

Před vlastním spuštěním dotazu pro něj obvykle „připravíme“ podmínky pomocí metody SQL -> Prepare(). Za touto přípravou se skrývá lokace prostředků potřebných k provedení dotazu a uložení jeho výsledků. Příprava může urychlit průběh dotazu.

Vraťme se nyní k našemu zadání — zobrazovat vybraná pole z tabulek seřazená podle různých klíčů. Dotaz, který chceme položit, bude

```
select Jméno, P   jmen , M sto, Stav from zakaznik, ucty
where zakaznik.cis_zak =  ucty."Číslo zákazníka"
order by …
```

kde tečky nahrazují klíč, podle kterého budeme třídit.

Přidáme tedy do vizuálního návrhu našeho programu komponentu RadioGroup, kontejner pro skupinu přepínačů, a do ní vložíme tři přepínače s nápisy Příjmení, Bydliště a Stav účtu, které pojmenujeme rb_prijmeni, rb_jmeno a rb_stav. U posledního z nich nastavíme hodnotu vlastnosti Checked na true.

Vedle nich umístíme tlačítko s nápisem Zobraz, které způsobí, že se vytvoří dotaz podle zvoleného přepínače a zobrazí se jeho výsledek. Handler, který bude reagovat na stisknutí tohoto tlačítka, bude mít tvar

```
void __fastcall THlavniOkno::tl_zobrazClick(TObject *Sender)
{
 TQuery * Q = DataModule1 -> dotaz1;   // pomocné proměnné pro zkrácení zápisu
 TStrings *S = Q -> SQL;
 Q -> Close();                         // Deaktivuj dotaz
 S -> Clear();                         // Vymaž předchozí a sestav nový
 S -> Add("select Jméno, Příjmení, Město, Stav from zakaznik, ucty");
 S -> Add("where");
 S -> Add("zakaznik.cis_zak =  ucty.\"Číslo zákazníka\"");
 S -> Add("order by ");
 if(rb_prijmeni->Checked)              // Část ORDER BY se liší podle
 {                                                   // zaškrtnutého přepínače
  S->Add("Příjmení");
 }
 else if(rb_bydliste->Checked)
 {
  S->Add("Město");
 }
 else
 {
  S->Add("Stav");
 }
 Q -> Prepare();                       // Připrav dotaz
 Q -> Open();                          // a spusť ho
}
```

Vytvoření dotazu a jeho provedení (soubor okno1.cpp)

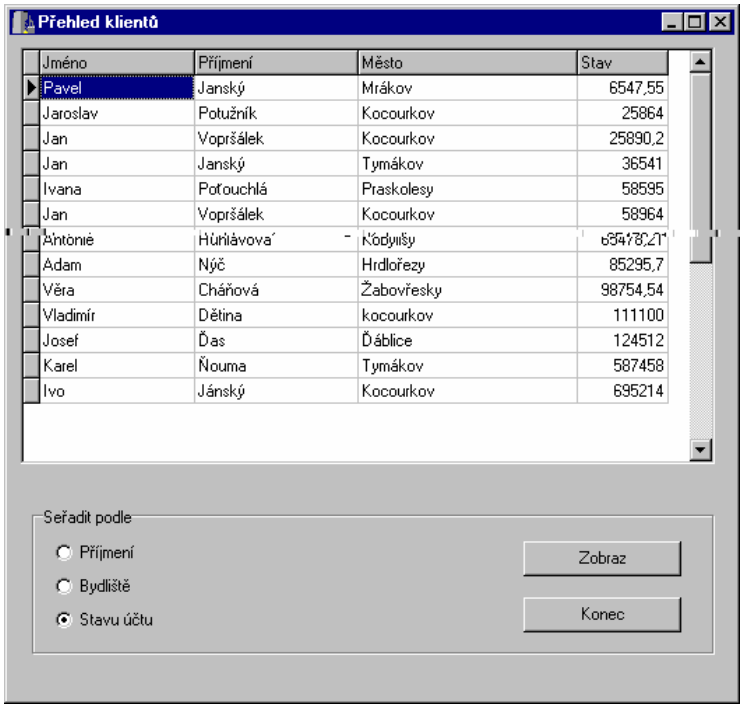
Aby tento program správně fungoval, musíme ještě zabezpečit, že se tento handler zavolá i na počátku. Zde si ale musíme pečlivě rozvážit, kdy a kde. Nelze ho volat v konstruktoru okna, neboť okno se vytváří před datovým modulem, a proto v době, kdy běží tělo konstruktoru okna, neexistují ještě komponenty v datovém modulu.

Musíme ho proto zavolat v konstruktoru datového modulu. Datový modul se vytváří až po hlavním okně a tělo jeho konstruktoru probíhá až po vytvoření všech komponent, které jsme do něj umístili, takže se nemusíme obávat problémů. Zde je výpis:

```
__fastcall TDataModule1::TDataModule1(TComponent* Owner)
    : TDataModule(Owner)
{ HlavniOkno->tl_zobrazClick(this);
}
```

Konstruktor datového modulu (soubor dm1.cpp)

Výsledek ukazuje obr. 8.7.



Obr. 8.7 Výpis databáze setříděné podle různých kritérií

Jestliže zvolíme některý z přepínačů a stiskneme Zobraz, vypíše se databáze setříděná podle zvoleného sloupce.

Poznámky

- ✧ Metoda Open() nejprve vyvolá událost BeforeOpen, pak provede příkaz SELECT a vytvoří výsledkovou množinu a nakonec vyvolá událost AfterOpen.
- ✧ Komponenta Query může pracovat i s dalšími příkazy jazyka SQL; ovšem pouze příkaz SELECT vytváří výsledkovou množinu. Pro příkazy INSERT, DELETE a UPDATE ovšem používáme metodu ExecSQL().
- ✧ Hotový program v této podobě najdete na doprovodném CD v adresáři KAP08\01.

Parametry v SQL

Jazyk SQL umožňuje používat v mnoha situacích parametry — místo konkrétní hodnoty uvést v příkazu identifikátor, před kterým stojí dvojtečka. Za běhu pak lze za tyto parametry dosazovat.

Komponenta Query umožňuje pracovat s parametry prostřednictvím vlastnosti Params, která je přístupná jak v době návrhu tak i za běhu programu. My si ukážeme jednoduchý příklad, ve kterém budeme zjišťovat, kdo z klientů má na účtu více než jistou částku. Tuto částku budeme zadávat v editačním poli.

Dotaz v jazyce SQL bude jednoduchý a zadáme ho už v době návrhu programu:

```
SELECT jméno, p   jmen  , stav FROM zakaznik, ucty
WHERE stav > :stav AND zakaznik.Cis_zak = ucty."Číslo zákazníka"
ORDER BY stav
```

Zde :stav je parametr, jehož hodnotu zadáme za běhu programu.

Začneme nový projekt. Do vizuálního návrhu vložíme komponentu Query, kterou pojmenujeme Dotaz. Její vlastnosti DatabaseName přiřadíme hodnotu KLIENTI a do vlastnosti SQL uložíme výše uvedený dotaz. Komponenta sama už v době návrhu pozná, že :stav je parametr.

Dále sem vložíme komponentu DataSource, které ponecháme implicitní jméno DataSource1. Tato komponenta bude čerpat data z komponenty Dotaz (vlastnost DataSource). Nakonec přidáme komponentu DBGrid, která bude čerpat data z komponenty DataSource1 a tato data bude zobrazovat.

Dále vložíme do vizuálního návrhu komponentu Edit, kterou pojmenujeme ed_kolik, a komponentu Button, kterou pojmenujeme tl_otaznik. Celý program se bude skládat z handleru reagujícího na stisknutí tlačítka:

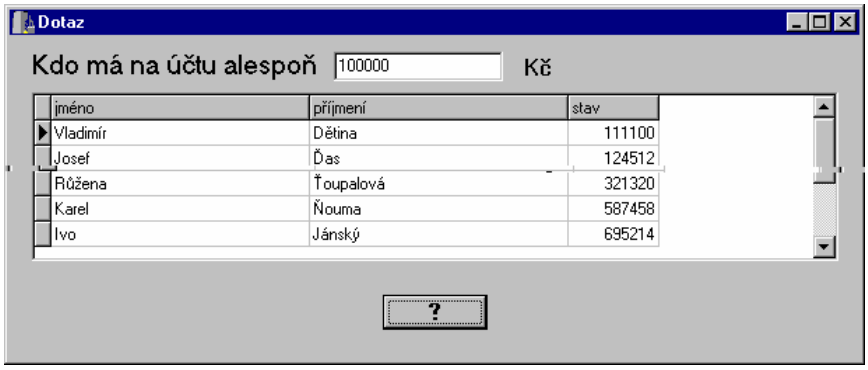
```
void __fastcall TForm1::tl_otaznikClick(TObject *Sender)
{
  Dotaz -> Active = false;
  Dotaz -> Params-> Items[0]-> AsInteger = ed_kolik->Text.ToInt();
  Dotaz-> Open();
}
```

Určíme hodnotu parametru a spustíme dotaz (soubor okno.cpp)

Program za běhu ukazuje obr. 8.8.

Poznámky

- ✧ Ne všechny součásti dotazu lze parametrizovat. Nelze parametrizovat např. jména tabulek. Přesná pravidla mohou záviset na použitém serveru nebo databázovém stroji.
- ✧ Tento prográmek najdete na doprovodném CD v adresáři KAP08\02.
- ✧ Je jasné, že tento program má řadu závažných nedostatků — např. nekontroluje, zda uživatel správně zadal hodnotu, na kterou se ptá. Chybné zadání může způsobit výjimku.



Obr. 8.8 Výsledky dotazu s parametry

8.3 Pohyby na účtech

Naším dalším úkolem bude napsat program, který vytvoří tabulku obsahující příkazy k převodům peněz mezi jednotlivými účty. Budeme předpokládat, že jde pouze o převody peněz mezi našimi zákazníky. Tato tabulka mít jméno zmenyN.db, kde N bude její pořadové číslo, a bude uložena spolu s ostatními ve stejném adresáři. (Jinými slovy, použijeme týž alias KLIENTI.) K zadávání údajů, které se budou ukládat do tabulky, použijeme „obyčejné“, tj. nikoli databázové komponenty.

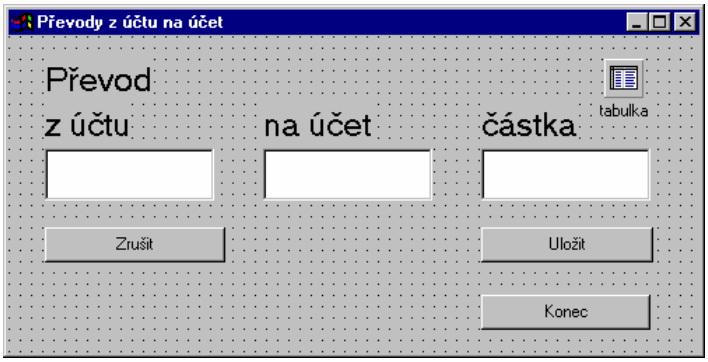
Jako obvykle začneme nový projekt, který uložíme pod jménem prevod. Hlavní okno dostane tradiční jméno HlavniOkno a bude uloženo v modulu okno. Do vizuálního návrhu okna vložíme tři komponenty Edit, které pojmenujeme ed_odkud, ed_kam a ed_kolik a které budou sloužit k zadávání čísla výchozího a cílového účtu a převáděné částky. K nim umístíme vysvětlující nápisy (komponenty Label, viz obr. 8.8).

Dále do tohoto okna přidáme tři tlačítka, tj. komponenty Button, které pojmenujeme tl_zrus, tl_konec a tl_uloz. Stisknutím prvního z nich zrušíme rozdělanou akci, tj. vymažeme data zapsaná v editačních polích, stisknutím druhého ukončíme program a stisknutím třetího data zapsaná ve vstupních polích uložíme do souboru.

Pak u těchto tří komponent nastavíme vlastnost TabOrder postupně na hodnoty 0 – 5 tak, abychom mohli přesunovat fokus pomocí klávesy TAB v pořadí zleva doprava nejprve na vstupní pole, pak na tlačítka.

Nakonec vložíme do vizuálního návrhu komponentu Table, kterou pojmenujeme vtipně tabulka. Do její vlastnosti DatabaseName uložíme alias KLIENTI, vlastnost TableName ponecháme prázdnou, ale vlastnosti TableType dáme hodnotu ttParadox (půjde o tabulku pro Paradox).

Hotový vizuální návrh našeho programu ukazuje obr. 8.9.



Obr. 8.9 Vizuální návrh programu pro vytvoření tabulky změn

Vytváříme tabulku za běhu programu

Abychom mohli novou tabulku vytvořit za běhu programu, musíme definovat její sloupce. To lze ale zvládnout ještě v době návrhu. Vybereme komponentu tabulka, v inspektoru objektů vyhledáme vlastnost FieldDefs a stisknutím tlačítka se třemi tečkami vedle její hodnoty vyvoláme editor sloupců (obr. 8.10). Stisknutím tlačítka Add postupně přidáme do tabulky tři sloupce, které prostředí pojmenuje tabulkaField0, tabulkaField1 atd. (Přesněji: V programu přibudou tři komponenty typu TFieldDef popisující jednotlivé sloupce naší databázové tabulky.)

Postupně v editoru sloupců každý ze sloupců vybereme a v inspektoru objektů jej pojmenujeme Odkud, Kam a Kolik (vlastnost Name).

U prvních dvou přiřadíme vlastnosti DataType hodnotu ftInteger (celá čísla), u posledního ftFloat (reálná čísla).[50] Tím definujeme typ odpovídajících sloupců, tj. typ hodnot, které se budou do odpovídajících sloupců zapisovat.

Nakonec nastavíme u všech tří polí vlastnost Required na true — databáze bude zadání těchto hodnot vyžadovat.

Obr. 8.10 Vytvoříme definice jednotlivých sloupců tabulky

Nyní se můžeme pustit do programování. Tabulku vytvoříme v konstruktoru okna. Nejprve sestavíme jméno zmeny0.db, přiřadíme ho vlastnosti tabulka->TableName a zjistíme, zda tabulka s tímto jménem existuje. K tomu použijeme metodu tabulka->Exists(). Pokud taková tabulka existuje, zkusíme jméno zmeny1.db atd., až najdeme první volné jméno. Jméno tabulky také připojíme do titulku okna.

Pak se pokusíme tabulku vytvořit. K tomu použijeme metodu CreateTable(). Pokud se tabulku nepodaří vytvořit, vznikne výjimka. V tom případě program ukončíme.

Mimo to vymažeme texty ve všech třech vstupních polích. Protože tuto akci budeme potřebovat na několika místech v programu, uložíme ji do samostatné metody vymazTexty().

Zdrojový kód konstruktoru okna a metody vymazTexty() ukazuje následující výpis.

```
__fastcall THlavniOkno::THlavniOkno(TComponent* Owner)
  : TForm(Owner)
{                                                    // Kontrola existence tabulky:
 String jm = "zmeny";                  // Základ jména souboru
 static int i = 0;                     // Počítadlo
 do {
  String jmeno = jm + i++ + ".db";     // Zkonstruuj jméno
```

[50] Protože jde o peníze, bylo by rozumnější použít typ ftCurrency.

```
   tabulka -> TableName = jmeno;          // A zkus, zda taková tabulka
  } while(tabulka -> Exists);             // existuje
  try{
   tabulka -> CreateTable();              // Pak ji zkus vytvořit.
   Caption = Caption + " - " + tabulka->TableName;
   vymazTexty();                                  // Vymaž nápisy v editačních polích
  }
  catch(...)                                              // Pokud se nepovede vytvořit tabulku,
  {                                                                  // upozorni uživatele
   Application -> MessageBox("Nepodařilo se vytvořit tabulku", "Problém", MB_OK);
   Application -> Terminate();            // a ukonči program.
  }
}

void THlavniOkno::vymazTexty()
{
   ed_kolik->Text = "";     // Vymaže texty ve všech třech editačních polích
   ed_kam ->Text = "";
   ed_odkud->Text = "";
}
```

Konstruktor okna vytvoří nové jméno a zkusí vytvořit tabulku (soubor okno.cpp)

Stisknutím tlačítka Konec ukončíme náš program, tj. zavoláme notoricky známou metodu Close(). Přitom bychom se ale měli postarat o uzavření databáze. Program však můžeme ukončit i jinými způsoby, a proto tuto akci svěříme handleru, který se volá při uzavírání okna, tedy při události OnClose.

```
void __fastcall THlavniOkno::tl_konecClick(TObject *Sender)
{
   Close();
}

void __fastcall THlavniOkno::FormClose(TObject *Sender, TCloseAction &Action)
{
   tabulka -> Active = 0;        // Uzavři tabulku
}
```

Ukončení programu a uzavření databázové tabulky (soubor okno.cpp)

Zbývá zpracovat vkládání záznamů do tabulky. O to se postaráme v metodě, která bude ošetřovat stisknutí tlačítka Uložit. Zde musíme nejprve tabulku otevřít tím, že vlastnosti Active přiřadíme hodnotu true. Pak ji převedeme do režimu úprav voláním metody Edit() a pomocí metody Insert() do ní vložíme nový prázdný záznam.

Dále vložíme do jednotlivých sloupců tabulky hodnoty z editačních polí formuláře. Pro přístup k hodnotám sloupců tabulky použijeme vlastnost Fields, která představuje všechny sloupce tabulky jako celek.[51] Vlastnost Fields tabulky má opět vlastnost Fields, která představuje pole hodnot jednotlivých sloupců. První dvě hodnoty ukládáme jako celá čísla, poslední jako reálné číslo; použijeme tedy vlastností AsInteger, resp. AsFloat.

Hodnoty zapsané do vlastnosti Fields uložíme do nového aktuálního záznamu v tabulce voláním metody Post(). Nakonec tabulku uzavřeme, vymažeme texty ve vstupních polích formuláře a přeneseme fokus na první z nich. Zde je úplný zdrojový text této metody:

```
void __fastcall THlavniOkno::tl_ulozClick(TObject *Sender)
{
   tabulka->Active=true;       // Otevřeme tabulku
   tabulka->Edit();            // Převedeme ji do režimu úprav
   tabulka->Insert();               // Vložíme nový prázdný záznam
                                                  // Přiřadíme sloupcům hodnoty
   tabulka->Fields->Fields[0]->AsInteger=ed_odkud->Text.ToInt();
   tabulka->Fields->Fields[1]->AsInteger=ed_kam->Text.ToInt();
   tabulka->Fields->Fields[2]->AsInteger=ed_kolik->Text.ToDouble();
   tabulka->Post();                 // Zapíšeme změny do souboru
   tabulka->Active=false;      // Uzavřeme tabulku
   vymazTexty();                    // Vymažeme vstupní pole formuláře
   FocusControl(ed_odkud);  // Přeneseme fokus na první vstupní pole
}
```

Ukládání hodnot do tabulky (soubor okno.cpp)

Hotový program v této podobě najdete na doprovodném CD v adresáři KAP08\03.

[51] Pozor, zde se liší C++ Builder 4 od předchozí verze. Viz poznámku na konci tohoto oddílu.

Poznámky

- ✧ V C++ Builderu 3 má tabulka (a podobně i další třídy odvozené od TDataSet) vlastnost Fields, která se přímo chová jako pole jednotlivých sloupců. To znamená, že příkazy, kterým přiřazujeme sloupcům hodnoty, budou mít tvar např.

```
tabulka->Fields[0]->AsInteger=ed_odkud->Text.ToInt();
```

- ✧ Pro tabulky zmenaN jsme nedefinovali žádný index, a proto se vytvoří pouze dva soubory (zmenyN.db a zmenyN.val).
- ✧ Kód pro vkládání hodnot do tabulek obsahuje dva možné zdroje běhových chyb. První z nich už známe: pokud uživatel zadá do editačního pole hodnotu, kterou nelze interpretovat jako číslo odpovídajícího typu, vznikne výjimka typu EConvertError. To lze vyřešit podobným způsobem jako v kapitole 6.2 (oddíl *Chybný vstup*).
- ✧ Druhý problém se skrývá ve skutečnosti, že v naší tabulce je zadání hodnot všech tří sloupců povinné. Pokud bychom některou z hodnot vynechali, vznikne výjimka EDatabaseError.

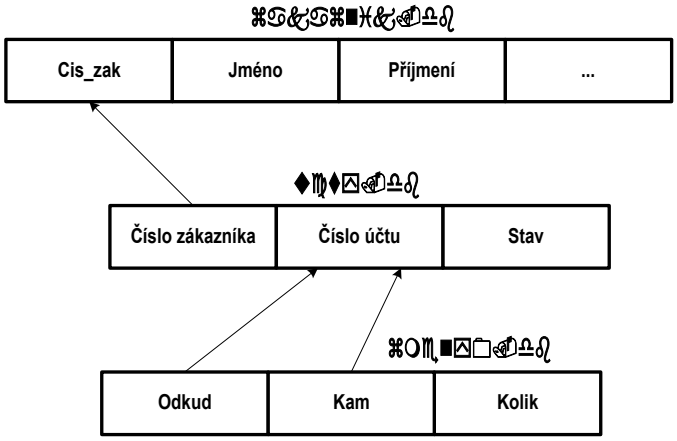
Prohlížíme a provádíme převody

Zbývá nám poslední úkol — napsat program, který umožní prohlédnout si vytvořený soubor s požadavky převodů a pak tyto převody provést.

Začneme prohlížením změn. Náš program bude postupně procházet jednotlivé záznamy s tabulce zmeny0 a u každého požadavku vypíše číslo výchozího účtu, jméno a příjmení majitele a stav účtu, tytéž informace o cílovém účtu a jeho majiteli a převáděnou částku. Potřebné informace budeme čerpat z tabulek zakaznik a ucty.

Vztahy mezi tabulkami naznačuje obrázek 8.11: V tabulce zmeny si ve sloupci Odkud nebo Kam přečteme číslo účtu, který nás zajímá. Pak prohledáme tabulku ucty, najdeme účet se stejným číslem, zjistíme jeho stav a číslo zákazníka. Podle čísla zákazníka vyhledáme potřebné údaje o majiteli v tabulce zakaznik.

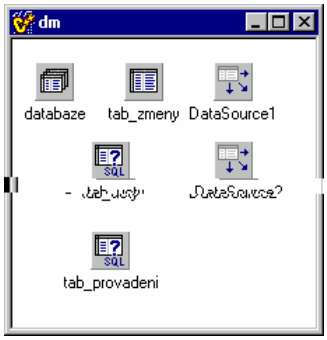
Toto prohlížení bychom mohli nejspíš zvládnout prostředky, které už známe — pomocí tabulek master-detail a filtrů. Použijeme však jiný postup, který nám umožní podrobněji se seznámit s fungováním komponenty Table a Query.



Obr. 8.11 Vztahy mezi použitými databázovými tabulkami

Vizuální návrh

Vytvoříme nový projekt, který uložíme pod jménem prevod. Hlavní okno pojmenujeme jako obvykle HlavniOkno a uložíme je do modulu okno. Dále vytvoříme nový datový modul, který uložíme pod výstižným názvem dm do souboru dm1. Do modulu okno vložíme hlavičkový souboru datového modulu dm a naopak, do datového modulu dm vložíme hlavičkový soubor okenního modulu okno.



Obr. 8.12 Vizuální návrh datového modulu

V našem programu budeme potřebovat několik tabulek napojených na stejný alias; vedle toho při provádění převodů budeme pracovat s transakcemi, a proto použijeme komponentu Database, kterou najdeme na paletě Data Access. Vložíme ji do datového modulu, pojmenujeme ji databaze, její vlastnosti DatabaseName přiřadíme alias KLIENTI a její vlastnosti TransIsolation dáme hodnotu tiDirtyRead. (Vlastnost TransIsolation určuje úroveň izolace v průběhu transakcí — např. zda mohou jiné programy v průběhu transakce číst hodnoty v záznamech, kterých se transakce týká. Pro databáze Paradox a dBase požaduje BDE úroveň tiDirtyRead.)

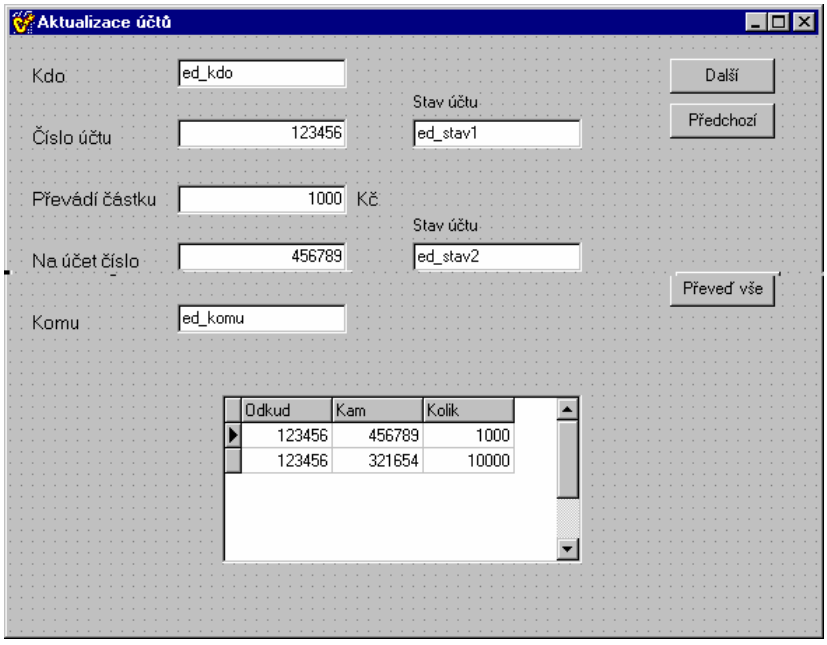
Dále vložíme do datového modulu komponentu Table, kterou pojmenujeme tab_zmeny. Tato komponenta bude sloužit k procházení databázové tabulky zmeny0. Její vlastnosti DatabaseName dáme hodnotu KLIENTI a vlastnosti TableName hodnotu zmeny0.db. Vlastnosti Active přiřadíme hodnotu true.

Pak vložíme do datového modulu dvě komponenty Query, které pojmenujeme tab_ucty a tab_provedeni. První z nich bude sloužit k prohlížení převodů, druhá k jejich provedení. U obou přiřadíme vlastnosti DatabaseName hodnotu KLIENTI.

Komponenta tab_provedeni bude sloužit k aktualizaci účtů, tedy k provádění převodních příkazů. Ovšem komponenta Query implicitně neumožňuje editovat výsledky dotazu; proto musíme ještě nastavit její vlastnost RequestLive na true.[52]

Nakonec přidáme dvě komponenty DataSource, kterým ponecháme implicitní jména DataSource1 a DataSource2. První z nich připojíme (vlastnost DataSet) ke komponentě tab_zmeny, druhou ke komponentě tab_ucty. Hotový datový modul ukazuje obr. 8.12.

Nyní se pustíme do vizuálního návrhu hlavního okna programu (obr. 8.13).



Obr. 8.13 Vizuální návrh hlavního okna programu

[52] Ani pak není zaručeno, že výsledky dotazu bude možno editovat. Přesné podmínky najdete v nápovědě; dotaz, který my použijeme, bude tyto podmínky splňovat.

Do tohoto okna nejprve vložíme komponentu DBGrid, které ponecháme implicitní jméno DBGrid1; napojíme ji na komponentu DataSource1, která čerpá data z tabulky tab_zmeny (vlastnosti DataSource přiřadíme dm->DataSource1). Bude zobrazovat obsah tabulky požadovaných změn zmeny0.

Pak přidáme tři komponenty DBEdit (editační databázové pole). Ponecháme jim implicitní jména, jejich vlastnosti DataSource přiřadíme dm->DataSource1 a jejich vlastnosti DataField přiřadíme postupně hodnoty Odkud, Kolik a Kam. Tím jsme určili, že tato pole budou zobrazovat hodnoty z odpovídajícího aktuálního sloupce tabulky zmena0.

Jména obou účastníků převodu a stav jejich účtů budeme zjišťovat dvěma dotazy (nebo přesněji jedním dotazem se dvěma různými hodnotami parametru). Proto není vhodné pole, která je budou zobrazovat, přímo napojit na zdroje dat; použijeme tedy komponenty Edit, které pojmenujeme ed_kdo, ed_komu, ed_stav1 a ed_stav2.

Nakonec přidáme tři tlačítka, která pojmenujeme tl_dalsi, tl_predchozi a tl_proved. První dvě budou sloužit k postupnému procházení souboru s převodními příkazy, třetí bude mít za úkol všechny převody provést.

Handlery pro první dvě tlačítka budou prostě volat metody Next(), resp. Prior(). Handler pro třetí z nich zavolá metodu GlobalniPrevod(), kterou definujeme v datovém modulu a která se postará o provedení všech převodních příkazů.

```
void __fastcall THlavniOkno::tl_dalsiClick(TObject *Sender)
{
  dm->tab_zmeny->Next();       // Přejdi na následující záznam
}
void __fastcall THlavniOkno::tl_predchoziClick(TObject *Sender)
{
  dm->tab_zmeny->Prior();  // Přejdi na předchozí záznam
}
void __fastcall THlavniOkno::tl_prevedClick(TObject *Sender)
{
 dm->GlobalniPrevod();
}
```

Obsluha tlačítek (soubor okno.cpp)

Zjišťujeme jméno a stav účtu

Nyní se ale musíme postarat, aby všechna pole v okně našeho programu zobrazovala stále správné hodnoty. To bude mít na starosti komponenta tab_ucty, která je zjistí dotazem do tabulek zakaznik a ucty. Ovšem kdy tento dotaz položit?

Jedna z možností je využít handlery, které ošetřují stisknutí tlačítek další, resp. Předchozí. Pak by ovšem nebylo jasné, jak zařídit, aby se správná data zobrazila také hned po spuštění programu.

Proto využijeme události AfterScroll tabulky tab_zmeny, která nastává vždy po přechodu na předchozí, resp. následující záznam nebo po otevření tabulky. Její handler bude mít tvar

```
void __fastcall Tdm::tab_zmenyAfterScroll(TDataSet *DataSet)
{
  odkud = tab_zmeny->FieldByName("Odkud")->Value;  // Ulož hodnoty sloupců
  kam = tab_zmeny->FieldByName("Kam")->Value;      // do pomocných proměnných
  kolik = tab_zmeny->FieldByName("Kolik")->Value;
  par z1 = UdajeOMajiteli(odkud.ToInt());          // Zjisti údaje o majiteli účtu
  par z2 = UdajeOMajiteli(kam.ToInt());
  HlavniOkno->ed_kdo -> Text = z1.Jmeno();         // a zapiš je do editačních
  HlavniOkno->ed_stav1 -> Text = z1.Stav();        // polí
  HlavniOkno->ed_komu -> Text = z2.Jmeno();
  HlavniOkno->ed_stav2 -> Text = z2.Stav();
}
```

Vždy po přechodu na nový záznam zjistíme údaje o majitelích obou účtů (soubor dm1.cpp)

Zde nejprve z tabulky zjistíme čísla obou účtů a převáděnou částku a uložíme je do proměnných odkud, kam a kolik (definovaných jako složky třídy datového modulu). Pro přístup k hodnotám uloženým v daném poli využijeme metodu FieldByName(), které předáme jako parametr znakový řetězec obsahující jméno pole.

Pak zavoláme funkci UdajeOMajiteli(), která nám zjistí jméno a příjmení majitele účtu s daným číslem a jeho stav. Výsledek vrátí jako strukturu typu[53] par obsahující řetězec (AnsiString) a reálné číslo. Nakonec zjištěné hodnoty uložíme do odpovídajících editačních polí.

Funkce UdajeOMajiteli() využívá komponentu tab_ucty. To je dotaz (Query) a jistě si vzpomenete, že jsme dosud nedefinovali jeho vlastnost SQL, tedy text dotazu. Učiníme tak nyní. Vybereme komponentu tab_ucty, v inspektoru objektů vyhledáme vlastnost SQL a stisknutím tlačítka se třemi tečkami napravo od její hodnoty otevřeme editor dotazů.

Dotaz, kterým zjistíme potřebné údaje na základě čísla účtu, má tvar

```
select ucty."  slo z kazn ka", ucty."  slo   tu", stav, jméno, p   jmen
from ucty, zakaznik
where ucty."Číslo účtu" = :odkud
    and
    ucty."Číslo zákazníka" = zakaznik.Cis_zak
```

Tento dotaz obsahuje parametr :odkud. To znamená, že nejprve musíme dotaz uzavřít, nastavit hodnotu parametru, a pak můžeme dotaz otevřít. Z jeho výsledku pak zjistíme jméno a příjmení, spojíme je do jednoho řetězce a uložíme do pomocné proměnné jm. Pak zjistíme stav účtu, výsledek uložíme do pomocné proměnné stav a vrátíme pár vytvořený ze jména a stavu.

```
Tdm::par Tdm::UdajeOMajiteli(int cislo_zakaznika)
{
   tab_ucty -> Close();                                  // Uzavři a připrav parametr
   tab_ucty -> Params -> Items[0] -> AsString = String(cislo_zakaznika);
   tab_ucty -> Open();                                   // Otevři (polož dotaz)
   String jm = tab_ucty->FieldByName("Jméno")-> AsString
                        + " " + tab_ucty->FieldByName("Příjmení")-> AsString;
   double stav = tab_ucty->FieldByName("stav")-> AsFloat;
   return par(jm, stav);                                 // Ze zjištěných hodnot vytvoř
}                                                                  // p  r a vra   ho jako v sledek
```

Získáváme údaje o majiteli účtu s daným číslem (soubor dm1.cpp)

Funkce FieldByname() vrací hodnotu typu TField, což je Variant — typ definovaný jako unie většiny běžných datových typů. Pro práci s ní můžeme použít vlastností AsInteger, AsString atd., které poskytují výsledek odpovídajícího typu.

Na závěr tohoto oddílu si ukážeme ještě deklaraci třídy datového modulu Tdm.

```
class Tdm : public TDataModule
{
__published:  // IDE-managed Components
   TDatabase *databaze;
   TTable *tab_zmeny;
   TDataSource *DataSource1;
   TDataSource *DataSource2;
   TQuery *tab_ucty;
   TQuery *tab_provadeni;
   void __fastcall tab_zmenyAfterScroll(TDataSet *DataSet);
private:         // User declarations
   struct par              // Pomocná struktura pro uložení zjištěných informací
   {
    String jmeno;
    double stav;
    par(String j, double d): jmeno(j), stav(d){}
    String Jmeno() { return jmeno; }
    double Stav() { return stav; }
   };
   String odkud, kam;                    // Čísla obou účtů
   double kolik;                              // Převáděná částka
   par UdajeOMajiteli(int cislo_zakaznika);
public:               // User declarations
   __fastcall Tdm(TComponent* Owner);
   void GlobalniPrevod();          // Funkce pro zpracování všech převodních
};                                                    // p   kaz   najednou
```

[53] Typ par jsme definovali ve třídě Tdm datového modulu. Mohli jsme ale použít standardní šablonu pair<> definovanou ve standardní šablonové knihovně jazyka C++.

Deklarace typu datového modulu (soubor dm1.h)

Vlastní převod

V této chvíli náš program již umí procházet souborem s požadavky na převody a zobrazovat potřebné údaje; zbývá naprogramovat vlastní převody, tedy metodu GlobalniPrevod(). Jejím základem bude cyklus, ve kterém projdeme postupně všechny záznamy v tabulce zmeny0. K tomu můžeme použít cyklus

```
for(tab_zmeny -> FindFirst();    // Najdi prvn  z znam
   !tab_zmeny -> Eof;            // Už jsi zpracoval poslední?
   tab_zmeny -> Next()       // Přejdi na další
   ) { /* ... */ }
```

neboť

- ✧ metoda FindFirst() nás přesune na první záznam a tak zajistí inicializaci cyklu,
- ✧ vlastnost Eof bude mít hodnotu true, jestliže se pokusíme číst záznam za posledním záznamem v tabulce nebo jestliže se přesuneme na poslední záznam voláním metody FindLast(), takže ji lze použít jako podmínku opakování,
- ✧ metoda Next() přečte následující záznam a přemístí na něj kurzor, takže se postará o reinicializaci cyklu.

Při provádění převodů musíme nejprve najít záznam o účtu příkazce a odečíst z něj převáděnou částku a pak musíme najít záznam o účtu příjemce a danou částku přičíst k němu přičíst. Tuto akce musí proběhnou obě, nebo nesmí proběhnout ani jedna, jinak by se peníze ztratily. Proto je obě uzavřeme do jednoho celku jako transakci.

O transakce se v knihovně VCL stará komponenta Database (proto jsme ji do našeho programu přidali). Transakci spustíme voláním metody StartTransaction(). Pokud se nějaká z akcí nepodaří, vznikne výjimka; proto uzavřeme celou transakci do bloku try/catch. Skončí-li blok try normálně, proběhne Commit(), jinak přejde řízení do handleru, který obsahuje volání metody Rollback().

Vlastní příkazy pro změnu záznamů v databázi budou využívat tab_provadeni. To je opět komponenta Query obsahující dotaz

```
select stav from ucty where ucty."  slo   tu" = :cu
```

s parametrem :cu, který bude obsahovat číslo účtu. Všimněte si, že zde si vybíráme jediný sloupec obsahující stav účtu.

Nejprve musíme komponentu tab_provadeni uzavřít, připravit parametr (číslo účtu příkazce), pak ji otevřeme a převedeme do režimu editování. Poté vypočteme novou hodnotu pole a uložíme ji (voláním metody Post()).

Pak ji uzavřeme a uděláme totéž s číslem účtu příjemce. Nakonec ukončíme transakci.

Pokud to naprogramujeme, zjistíme, že náš program má dvě nevýhody:

- ✧ Je-li soubor se změnami delší, uvidíme v komponentách, které zobrazují data, míhat se různé hodnoty. To je způsobeno procházením tabulky tab_zmeny.
- ✧ Po skončení budou komponenty v okně programu ukazovat jiný záznam než při stisknutí tlačítka Provést.

Obojí je způsobeno procházením tabulky tab_zmeny. Řešení těchto problémů je jednoduché:

- ✧ Před spuštěním změn zavoláme metodu DisableControls(), která zabrání zobrazování výsledků v průběhu zpracování. Po ukončení zavoláme EnableControls(), která zobrazování výsledků zase povolí.
- ✧ Před spuštěním cyklu, ve kterém procházíme tabulku příkazů, vytvoříme pomocí metody GetBookmark() záložku (Bookmark) na aktuální záznam. Po skončení se vrátíme na tuto pozici voláním metody GotoBookmark.

Úplný výpis metody GlobalniPrevod() vypadá takto:

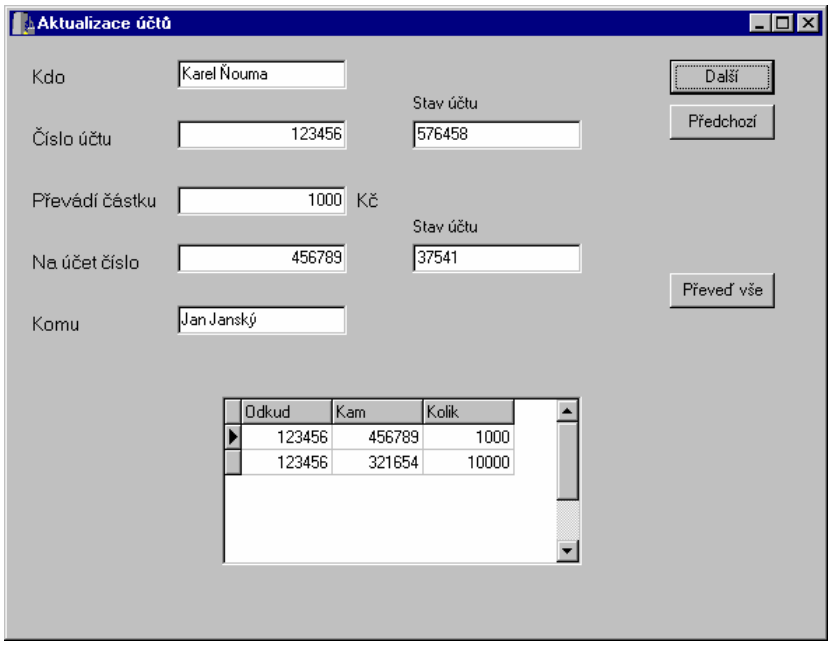
```
void Tdm::GlobalniPrevod()
{
 tab_zmeny -> DisableControls();                    // Aby nebylo vidět průběh
 tab_ucty -> DisableControls();
 TBookmark zde = tab_zmeny->GetBookmark();          // Záložka - sem se pak vrátíme
 for(tab_zmeny->FindFirst(); !tab_zmeny->Eof; tab_zmeny->Next())
 {
   databaze -> StartTransaction();                  // Vše nebo nic
```

```
  try{
    tab_provadeni -> Close();                    // Uzavři a připrav parametr
                                                                  // odkud připraveno v AfterScroll
    tab_provadeni -> Params -> Items[0] -> AsString = String(odkud);
    tab_provadeni -> Open();                     // Polož dotaz a začni editovat
    tab_provadeni -> Edit();                     // výsledek dotazu; změň hodnotu
                                                                  // Zde změníme hodnotu sloupce
    tab_provadeni -> Fields->Fields[0]->AsFloat -= kolik;
    tab_provadeni -> Post();                     // Ulož výsledky
    tab_provadeni -> Close();                    // Uzavři a připrav nový parametr
    tab_provadeni -> Params -> Items[0] -> AsString = String(kam);
    tab_provadeni -> Open();                     // a postupuj stejne jako prve
    tab_provadeni -> Edit();
    tab_provadeni -> Fields->Fields[0]->AsFloat += kolik;
    tab_provadeni -> Post();
    databaze -> Commit();                        // Je-li vše v pořádku, ulož
  }
  catch(...)
  {
    databaze-> Rollback();                       // jinak vše vrať
  }
 }
  tab_zmeny -> EnableControls();                 // Povol zobrazování výsledků
  tab_ucty -> EnableControls();
  tab_zmeny -> GotoBookmark(zde);                // a vrať se k zapamatované pozici
}                                                                 // v tabulce zm n
```

Metoda pro převody mezi účty (soubor dm1.cpp)

Poznámky

- ✧ Běžící program ukazuje obr. 8.14. Najdete ho na doprovodném CD v adresáři KAP08\04.
- ✧ Pro přístup k hodnotám polí jsme použili vlastnost Fields. Pozor na rozdíl mezi C++ Builderem verze 4 a předchozími verzemi (viz poznámku na konci oddílu *Vytváříme tabulku za běhu*).
- ✧ Tento program má řadu nedostatků; neumožňuje např. pracovat s více soubory se změnami s různými jmény, používá nevhodný typ pro zobrazení měny atd. Můžete zkusit tyto nedostatky odstranit.



Obr. 8.14 Program za běhu

9. Webovský prohlížeč Nic Moc

V této kapitole se krátce podíváme na programování pro internet a napíšeme si jednoduchý webovský prohlížeč neboli browser značky Nic Moc. I když se může zdát, že to je nošení dříví do lesa, neboť prohlížečů je na trhu k disposici dostatek, může takový program mít svůj význam. Stačí si např. představit, že chceme z jakéhokoli důvodu omezit přístup uživatelů k jistým adresám, třeba proto, aby v pracovní době dělali to, zač je platíme.

9.1 Komponenty

Než se pustíme do programování prohlížeče, zastavíme se alespoň stručně u komponent, které lze pro programování pro síť využít. Najdeme je na paletě Internet.

Vedle komponenty Powersock, která zapouzdřuje Winsock API, a komponent ServerSocket a ClientSocket, které umožňují programování na „nižší“ úrovni, zde najdeme komponenty, které umějí různé běžně používané operace při práci s WWW. Část těchto komponent jsou ve skutečnosti prvky ActiveX od firmy NetMasters; ty uvádí následující přehled.

NMDayTime	Od internetového časového serveru si vyžádá datum a čas.
NMEcho	Pošle text serveru echo a přijme text od něj.
NMFinger	Od internetového serveru finger si vyžádá informace o uživateli.
NMFTP	Slouží k přenosu souborů po síti prostřednictvím služby FTP.
NMHTPP	Slouží k přenosu dat prostřednictvím protokolu HTTP.
NMMsg	Posílá krátké zprávy jako ASCII text prostřednictvím protokolu TCP/IP.
NMMsgServ	Přijímá zprávy odeslané komponentou NMMsg.
NMNNTP	Odesílá a přijímá zprávy prostřednictvím protokolu NNTP.
NMPOP3	Vybere poštu, tedy zprávy uložené na poštovním serveru, prostřednictvím protokolu POP3.
NMUUProcessor	Zakóduje soubory v kódování MIME nebo UUENCODE nebo je rozkóduje.
NMSMTP	Odešle poštu prostřednictvím SMTP serveru.
NMStrm	Odešle datový proud síťovému nebo internetovému proudovému serveru.
NMStrmServ	Přijme datový proud odeslaný komponentou NMStrm.
NMTime	Od internetového časového serveru si vyžádá čas.
NMUDP	Přenáší data po síti pomocí protokolu UDP.
NMGeneralServer	Obecný TCP/IP server.
HTML	Základ browseru.
NMURL	Převede URL na znakový řetězec a naopak.
WebDispatcher	Předává požadavky ve formátu HTTP komponentám typu Action.

Vedle toho je na paletě Internet i několik skutečných vizuálních komponent, tedy součástí knihovny VCL.

PageProducer	Vytvoří posloupnost HTML příkazů podle zadané vstupní šablony.
QueryTableProducer	Sestaví posloupnost HTML příkazů, která zobrazí výsledek dotazu ve tvaru tabulky.
DataSetTableProducer	Sestaví posloupnost HTML příkazů, která zobrazí záznamy z datové množiny dotazu ve tvaru tabulky.
TDataSetPageProducer	Vytvoří posloupnost HTML příkazů podle zadané vstupní šablony.

9.2 Prohlížeč

Z předchozího přehledu je zřejmé, že při programování prohlížeče bude nejvhodnější využít práce, kterou za nás už někdo udělal, a použít komponentu HTML.

První verze

Vytvoříme tedy nový projekt, který uložíme pod jménem browser. Hlavní okno pojmenujeme jako obvykle HlavniOkno a uložíme ho do modulu okno.

Do tohoto okna — tedy do vizuálního návrhu naší aplikace — vložíme komponentu Panel. Její vlastnosti Align dáme hodnotu alTop, její vlastnosti Height přiřadíme hodnotu 60 a vlastnost Caption vymažeme. Panel bude zaujímat horní část klientské plochy okna.

Na tento panel položíme komponentu ComboBox,[54] kterou pojmenujeme com_URL, neboť v ní budeme zadávat nebo vybírat internetovské adresy (URL). Umístíme ji do horní části panelu, jeho šířku nastavíme přes celý panel. Do vlastnosti Text vložíme svou oblíbenou webovskou adresu, např. http://www.seznam.cz.

Dále přidáme do vizuálního návrhu okna (mimo panel) komponentu StatusBar, kterou pojmenujeme sb_stav. Její vlastnosti SimplePanel dáme hodnotu true.

Nakonec do okna umístíme komponentu HTML, pojmenujeme ji html_okno a její vlastnosti Align přiřadíme hodnotu alClient, takže vyplní celou zbývající uživatelskou oblast. Tato komponenta bude zobrazovat stažené HTML stránky. Aktuální stav vizuálního návrhu okna ukazuje obr. 9.1.

Ke stažení webovské stránky ze zadané adresy nám poslouží metoda RequestDoc() komponenty HTML, které jako parametr zadáme adresu (URL) ve tvaru znakového řetězce.

Nyní bychom si měli rozmyslet, jak stahování webovské stránky spustíme. Jednou možností je stisknutí klávesy ENTER, jinou možnost představuje kliknutí myší na zadané adrese. Později na panel přidáme také tlačítko, kterým budeme přenos spouštět.

Začneme ošetřením události OnClick komponenty com_URL. Bude nesmírně jednoduchý: Zkontroluje, zda je v kombu zadán nějaký text, a pokud ano, předáme ho metodě RequestDoc() jako parametr.

```
void __fastcall THlavniOkno::com_URLClick(TObject *Sender)
{
  if(com_URL->Text != "")
    html_okno->RequestDoc(com_URL->Text);
}
```

Reakci na stisknutí klávesy ENTER musíme napsat jako handler ošetřující událost OnKeyPress. Tento handler má jako druhý parametr Key znak odpovídající klávese, kterou uživatel stiskl.[55] Klávesu ENTER označuje předdefinovaná hodnota VK_RETURN.

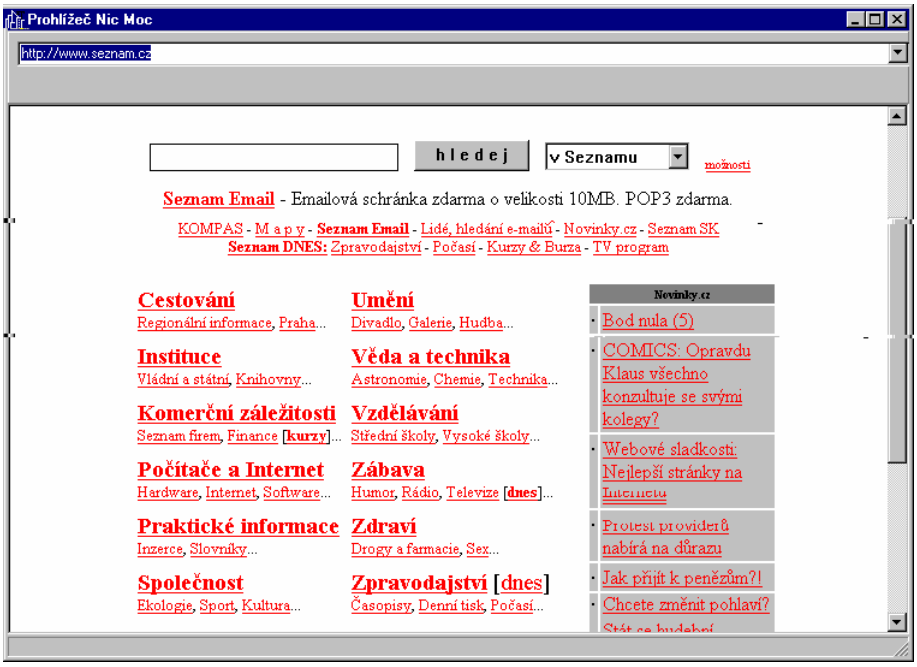
Náš handler tedy zkontroluje, zda uživatel stiskl klávesu ENTER. Pokud ano, vynuluje parametr Key, zkontroluje, zda je zadána nějaká adresa, uloží ji do seznamu a zavolá handler, který ošetřoval kliknutí myší.

```
void __fastcall THlavniOkno::com_URLKeyPress(TObject *Sender, char &Key)
{
  if(Key == VK_RETURN)                    // Je to Enter?
  {
   Key = 0;                               // Pokud ano, vynuluj parametr
   if(com_URL->Text == "") return;    // Je zadána adresa?
   com_URL->Items -> Insert(0, com_URL->Text);  // Zapamatuj si ji
   com_URLClick(Sender);                  // a začni stahovat dokument
  }
}
```

Stisknutím klávesy Enter odstartujeme stahování stránky (soubor okno.cpp)

[54] Připomeňme si, že kombo je ovládací prvek z Windows, který v sobě kombinuje editační pole a rozbalovací seznam. Vstupní hodnotu můžeme buď vybrat v seznamu nebo zapsat do editačního pole. Do seznamu budeme ukládat minulé adresy, abychom se k nim mohli snadno vrátit.

[55] Pokud bychom chtěli zjišťovat, zda byl zároveň stisknut také některý z přeřaďovačů (CTRL, ALT atd.), museli bychom použít událost OnKeyDown nebo OnKeyUp.



Obr. 9.1 Prohlížeč v akci

Přeložením a spuštěním se přesvědčíte, že náš prohlížeč je již provozuschopný, i když opravdu dělá čest svému jménu (nic moc). V této podobě ho najdete na doprovodném CD v adresáři KAP09\01.

Drobná vylepšení

Už jsme si slíbili, že na panel prohlížeče přidáme tlačítko, kterým budeme spouštět stahování zadané adresy. Vedle toho bychom uvítali možnost zastavit stahování, pokud přenos trvá příliš dlouho, a spustit ho znovu. Mnohé prohlížeče umožňují zobrazit zdrojový text stránky v jazyce HTML.

Nebylo by také špatné, kdybychom mohli ve stavové řádce sledovat postup stahování.

Začneme tlačítky. Umístíme na panel čtyři komponenty Button, které pojmenujeme postupně tl_jedeme, tl_stop, tl_znovu a tl_zdroj a umístíme na ně odpovídající nápisy (obr. 9.2).

Nyní vybereme tlačítko s nápisem Jedeme, v inspektoru objektů přejdeme na kartu Events a události OnClick přiřadíme handler com_URLClick (týž, který se stará o reakci na stisknutí klávesy ENTER). Stejný handler se bude starat i o reakci na stisknutí tlačítka Znovu — prostě pošleme znovy požádáme o stažení dokumentu z aktuální adresy.

Stisknutí tlačítka Stop představuje žádost o zrušení akce, o předčasné ukončení stavování. K tomu nám poslouží metoda Cancel(); přitom do stavové řádky vložíme text Hotovo (nebo jakýkoli jiný, který vám připadá v této situaci vhodný):

```
void __fastcall THlavniOkno::tl_stopClick(TObject *Sender)
{
  html_okno->Cancel(0);
  sb_stav -> SimpleText = "Hotovo";
}
```

Tlačítko Zdroj má za úkol přepnout prohlížeč do režimu zobrazení zdrojového textu. I tentokrát bude programování jednoduché: Způsob zobrazení je určen vlastností ViewSource typu bool. Stačí tedy nahradit její hodnotu hodnotou opačnou. Přitom zároveň změníme text na tlačítku tak, aby v případě, že prohlížeč zobrazuje zdrojový text, nabízel zobrazení dokumentu a naopak.

```
void __fastcall THlavniOkno::tl_zdrojClick(TObject *Sender)
{
   html_okno -> ViewSource = !html_okno -> ViewSource;
   if(html_okno -> ViewSource)
     tl_zdroj->Caption = "Dokument";
   else
     tl_zdroj->Caption = "Zdroj";
}
```

Přepnutí mezi zobrazením zdrojového textu a dokumentu (soubor okno.cpp)

Nakonec se postaráme o vypisování zpráv o průběhu stahování dokumentu.

Při odeslání požadavku na stažení stránky nastane událost OnDoRequestDoc. V tom okamžiku vypíšeme zprávu, že hledáme spojení se zadanou adresou.

```
void __fastcall THlavniOkno::html_oknoDoRequestDoc(TObject *Sender,
    const WideString URL, HTMLElement *Element, DocInput *DocInput,
    WordBool &EnableDefault)
{
  sb_stav->SimpleText = "Hledám spojení s " + URL + "...";
}
```

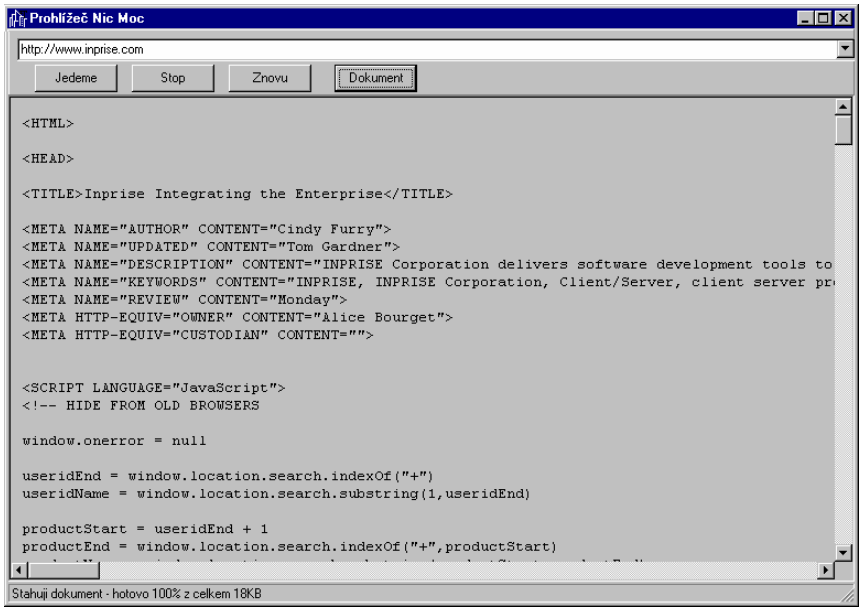
V okamžiku, kdy se naváže spojení se serverem a začne stahování dokumentu, nastane událost OnBeginRetrieval. V handleru, který na tuto událost reaguje, vypíšeme do stavové řádky text Spojeno a uložíme aktuální URL do seznamu:

```
void __fastcall THlavniOkno::html_oknoBeginRetrieval(TObject *Sender)
{
  sb_stav -> SimpleText = "Spojeno";
  com_URL->Items->Insert(0, com_URL->Text);
}
```

Rozsáhlejší dokumenty přicházejí po částech. Příchod každé takové části vyvolá událost OnUpdateRetrieval. V odpovídajícím handleru vypočteme, kolik procent již bylo přeneseno, a výsledek zapíšeme do stavové řádku. K tomu potřebujeme vědět, že celkový počet přenášených bajtů je uložen ve vlastnosti RetrieveBytesTotal a počet už přenesených bajtů je uložen ve vlastnosti RetrieveBytesDone. Poznamenejme ale, že stažení jedné stránky může znamenat přenos značného množství souborů, a protože informace o postupu se opírá o informace po přenosu jednotlivých souborů, slouží spíše k uklidnění uživatele než k odhadu doby potřebné k dokončení operace.

```
void __fastcall THlavniOkno::html_oknoUpdateRetrieval(TObject *Sender)
{
  int celk = html_okno->RetrieveBytesTotal;
  int hotovo = html_okno->RetrieveBytesDone;
  int proc = 0;
  if(celk && hotovo)
          proc = (100*hotovo)/celk;
  sb_stav -> SimpleText = String("Stahuji dokument - hotovo ")
                  + proc + " z celkem " + celk/1024 + "KB";
}
```

Program v této podobě najdete na doprovodném CD v adresáři KAP09\02. Obrázek 9,2 ukazuje tento prohlížeč při zobrazení zdrojového textu přenášené stránky.



Obr. 9.2 Vylepšený prohlížeč zobrazující zdrojový text

10. Opět hodiny, tentokrát jako komponenta

Ve 3. kapitole jsme napsali tolik verzí hodin, že je nejspíš už nemůžete ani vidět. Abyste se k nim nemuseli už nikdy vracet, uděláme si je teď naposledy, a to jako vizuální komponentu, kterou v zápětí instalujeme na novou paletu a použijeme v programu.

10.1 Jak se programují komponenty

Při programování komponent bohužel ztratíme většinu z výhod vizuálního programování. Navíc zde potřebujeme také znát rozšíření jazyka C++, která jsou k disposici v C++ Builderu. (Jejich přehled najdete v následující kapitole. Než budete pokračovat, doporučuji přečíst si alespoň část věnovanou vlastnostem (property).)

Asi nejčastěji postupujeme při vytváření nových komponent tak, že najdeme už hotovou komponentu, která dělá alespoň přibližně to, co potřebujeme, a upravíme ji — odvodíme od ní novou komponentu s potřebnými vlastnostmi. Chceme-li vytvořit vizuální komponentu založenou na oknu, můžeme použít společného předka TWinControl (od ní jsou odvozena např. tlačítka, editační pole atd.). Pokud nepotřebujeme, aby se naše komponenta chovala jako okno (nebude mít identifikační číslo, tzv. handle), můžeme použít předka TGraphicControl. Univerzálním předkem pro všechny komponenty, tj. i pro nevizuální, je třída TControl.

Náš postup je tedy následující:

- ✧ Vytvoříme samostatný modul (unit) obsahující definici třídy komponenty.
- ✧ Vložíme komponentu do balíčku[56] a přeložíme balíček i s komponentou.
- ✧ Instalujeme ji na zvolenou paletu.

Při programování komponent musíme mít na paměti několik pravidel:

- ✧ Komponenta je třída, která je nezávislá na zbytku programu a komunikuje s ním prostřednictvím svého rozhraní tvořeného vlastnostmi (property) a metodami.
- ✧ Předkem komponenty je třída TComponent nebo některá z tříd od ní odvozená.
- ✧ Instance komponent (a vůbec všech tříd odvozených od třídy TObject) musíme alokovat dynamicky.
- ✧ Vlastnosti, které se mají objevit v inspektoru objektů, musí být deklarovány jako publikované (sekce s přístupovými právy __published).

Při programování modulu s komponentou můžeme postupovat „ručně“, psát zdrojový text řádku po řádce. Můžeme ale také použít služeb komponentového šamana (Component Wizard). My si ukážeme druhou možnost, vysvětlíme si ale vše, co náš šaman vytvoří.

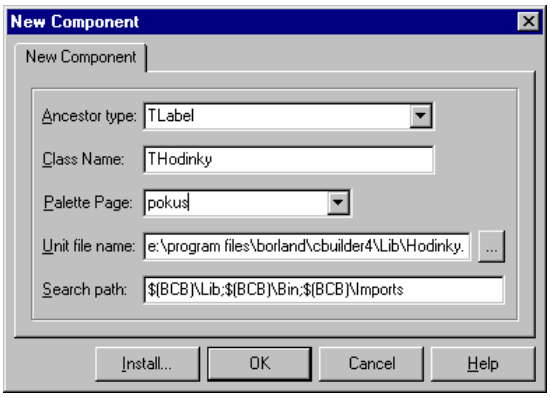
10.2 Začínáme

Nejprve si rozmyslíme, kterou třídu použijeme jako předka. Abychom si zjednodušili situaci, vytvoříme digitální hodiny — to znamená zobrazení textu. V úvahu tedy přicházejí např. komponenty Label nebo Panel. Použijeme první z nich.

Nyní vytvoříme modul s definicí komponenty. Příkazem File Close All... uzavřeme všechny otevřené projekty a soubory a příkazem Component | New Component... vyvoláme dialogové okno New Component (obr. 10.1).[57] V něm do pole Ancestor type vyplníme jméno předka (TLabel), do pole Class Name jméno třídy komponenty (THodinky) a do pole Palette page jméno palety, na kterou chceme novou komponentu instalovat. Zadáme-li jméno neexistující palety, při instalaci komponenty se vytvoří.

[56] Balíček, package, je zvláštní typ dynamické knihovny používaný v Borland C++ Builderu a v Delphi. Může obsahovat jednu nebo několik komponent.

[57] Stejného výsledku dosáhneme, vyvolám-li zásobník objektů příkazem File | New... a vybereme-li si ikonu Component.



Obr. 10.1 Vytváříme novou komponentu

V poli Unit file name zvolíme adresář, do kterého chceme vytvořený zdrojový soubor uložit. V C++ Builderu 4 zvolíme adresář …\Projects\Lib, v C++ Builderu 3 …\Projects. Pokud není tento adresář uveden i v následujícím poli, Search path, připíšeme ho tam, neboť to je cesta, kde bude C++ Builder hledat další moduly potřebné při překladu balíčku. Pak stiskneme tlačítko OK.

Prostředí vytvoří zdrojový a hlavičkový soubor pojmenovaný podle třídy komponenty — v našem případě to bude Hodinky.cpp a Hodinky.h. V hlavičkovém souboru najdeme kostru definice třídy Thodinky:

```
class PACKAGE THodiny : public TLabel
{
private:
protected:
public:
  __fastcall THodiny(TComponent* Owner);
__published:
};
```

Makro PACKAGE se rozvine v modifikátor, který říká, že jde o třídu exportovanou z balíčku; je tedy nezbytné.

V souboru Hodinky.cpp najdeme vedle konstruktoru této třídy ještě funkci ValidCtrCheck(), která kontroluje, zda nově vytvořená třída komponenty neobsahuje nějakou čistě virtuální funkci, a funkci Register(), která se stará o registraci komponenty.

```
static inline void ValidCtrCheck(THodinky *)
{
  new THodinky(NULL);
}

__fastcall THodinky::THodinky(TComponent* Owner)
  : TLabel(Owner)
{}

namespace Hodinky
{
  void __fastcall PACKAGE Register()
  {
    TComponentClass classes[1] = {__classid(THodinky)};
    RegisterComponents("pokus", classes, 0);
  }
}
```

Při ukládání nám prostředí ještě nabídne možnost soubory přejmenovat.

Registrace

Pokud budeme psát komponenty ručně, můžeme funkci ValidCtrCheck() vynechat. Funkce Register() je však nezbytná. Musí mít prototyp

```
void __fastcall PACKAGE Register();
```

a musí ležet v prostoru jmen (namespace) se stejným jménem jako je jméno komponenty bez úvodního T (v našem případě tedy Hodinky). Může najednou registrovat i větší množství komponent.

V této funkci nejprve vytvoříme pole classes s ukazateli na tabulky virtuálních metod jednotlivých komponent příkazem

```
TComponentClass classes[1] = {__classid(THodinky)};
```

Ke získání ukazatele na tabulku virtuálních metod slouží operátor __classid. Pak zavoláme funkci RegisterComponents(), která se postará o vlastní registraci. Jejím prvním parametrem je znakový řetězec obsahující jméno palety, na který chceme novou komponentu umístit (v našem případě "pokus") druhým je adresa pole obsahujícího tabulky virtuálních metod a třetím je index posledního prvku pole classes.

Tuto funkci zavolá prostředí při instalaci balíčku automaticky.

Komponenta Hodinky

Komponenta vytvořená šamanem umí přesně totéž co její předek, komponenta Label — má stejné metody, stejné vlastnosti atd.. Nyní ji tedy naučíme zobrazovat čas. Základem bude opět časovač, tj. komponenta Timer, která bude vyvolávat události OnTimer. V handleru reagujícím na tyto události pak budeme měnit text zobrazovaný komponentou. Vše potřebné již známe ze 3. kapitoly, a proto můžeme postupovat rychle.

Časovač tentokrát vložíme do programu „ručně“, neboť prostředí nám v této situaci nepomůže. Do definice třídy dopíšeme jako soukromou složku ukazatel tajmr na typ TTimer a funkci OnTajmr(), která bude sloužit jako handler reagující na zprávy od časovače.

V sekci __published definujeme novou vlastnost, Enabled typu bool. Tato vlastnost bude určovat, zda hodinky poběží. Protože bude využívat vlastnosti Enabled časovače, musíme pro ni definovat také přístupové metody, které v pojmenujeme GetEnabled() a SetEnabled(). Její implicitní hodnota bude false. Upravená deklarace třídy THodinky tedy bude mít tvar

```
class PACKAGE THodiny : public TLabel
{
private:
  TTimer * tajmr;
  //__property Caption;
protected:
  void __fastcall OnTajmr(TObject *Sender);
  void __fastcall SetEnabled(bool);
  bool __fastcall GetEnabled(void);
public:
    __fastcall THodiny(TComponent* Owner);
__published:
  __property bool Enabled={read=GetEnabled, write=SetEnabled, default=false};
};
```

Deklarace třídy THodinky (soubor hodinky.h)

Nyní naprogramujeme jednotlivé metody. V konstruktoru komponenty alokujeme nový časovač a jeho vlastnosti Interval přiřadíme hodnotu 1000, tj. 1 vteřina. Pak určíme, že při události OnTimer se bude volat metoda OnTajmer(). (Připomeňme si, že jména událostí — např. OnTimer — představují ukazatele na metody. Stačí tedy přiřadit datové složce OnTimer adresu metody OnTajmr().)

Nakonec uložíme false do vlastnosti Enabled časovače. Tím vlastně nastavíme implicitní hodnotu vlastnosti Enabled komponenty Hodinky.

```
__fastcall THodinky::THodinky(TComponent* Owner)
    : TLabel(Owner)
{
  tajmr = new TTimer(this);     // Vytvoříme dynamicky časovač
  tajmr -> Interval = 1000; // Nastavíme interval událostí OnTimer
  tajmr -> OnTimer = OnTajmr;    // Určíme handler, který se bude při události
  Caption = "00:00:00";          // OnTimer volat, a počáteční nápis
  tajmr->Enabled=false;          // Na počátku časovač neběží
}
```

Konstruktor třídy THodinky (soubor hodinky.cpp)

Handler OnTajmr() musí být typu void __fastcall a musí mít jeden parametr typu TObject*. Jeho jediným úkolem je zjistit přesný čas, převést jej na řetězec a přiřadit vlastnosti Caption zděděné po předkovi, třídě TLabel. To už známe:

```
void __fastcall THodinky::OnTajmr(TObject *Sender)
{
  TLabel::Caption = Time();
}
```

Dále napíšeme přístupové metody SetEnabled() a GetEnabled(). Metoda, která nastavuje hodnotu dané vlastnosti, Set… musí být typu void a musí mít jeden parametr typu této vlastnosti. V našem případě tedy bude mít jeden parametr typu bool. Hodnotu tohoto parametru předá vlastnosti Enabled časovače.

Metoda Get…, která zpřístupňuje danou vlastnost, musí být bez parametrů a musí vracet hodnotu typu této vlastnosti; v našem případě tedy bude vracet hodnotu typu bool. Vrátí hodnotu vlastnosti Enabled časovače.

```
void __fastcall THodinky::SetEnabled(bool val)
{
 tajmr -> Enabled = val;
}

bool __fastcall THodinky::GetEnabled()
{
 return tajmr->Enabled;
 }
```

Přístupové metody pro vlastnost Enabled naší komponenty (soubor Hodinky.cpp)

Tím jsme hotovi, můžeme vytvořené soubory uložit a uzavřít. Takto vytvořenou komponentu můžeme používat podobně jako jakoukoli jinou třídu v programu a v této podobě ji také budeme ladit. (Nesmíme ale zapomenout, že jde o potomka třídy TObject, takže její instance smějí být jen dynamické.)

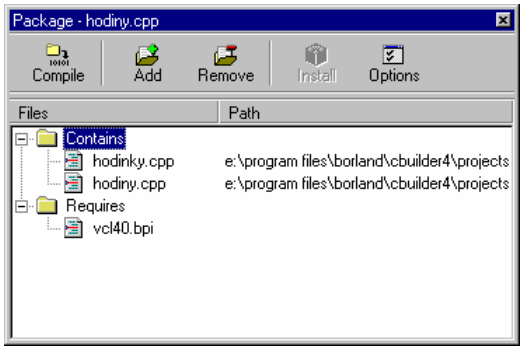
Poznámka

Všimněte si, že jsme se nikde nestarali o destrukci časovače a o jeho uvolnění z paměti. Nemusíme, neboť časovač je vlastněn naší komponentou (ukazatel na ni dostal jako parametr konstruktoru) a ta se o jeho zrušení postará sama.

Balíček

Nyní je třeba vytvořenou komponentu vložit do balíčku, přeložit, odladit a instalovat. Začneme vytvořením balíčku. Příkazem File | New... vyvoláme zásobník objektů a zvolíme v něm možnost Package (balíček). Prostředí vytvoří nový balíček a otevře okno Package, které obsahuje nástroj podobný správci projektů (obr. 10.2). Příkazem File | Save All... balíček uložíme a přitom mu dáme jméno Hodiny.

Povšimněte si, že tento „správce balíčků“ ukazuje nejen moduly, které balíček obsahuje (větev Contains), ale i moduly, na kterých závisí (větev Requires). (Pokud si toto okno jakýmkoli způsobem skryjete, můžete jej znovu vyvolat tak, že příkazem View | Window list nebo klávesovou zkratkou ALT+0 vyvoláte seznam všech otevřených oken C++ Builderu a v něm zvolíte Package.)



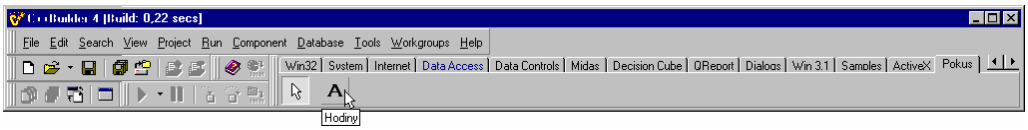
Obr. 10.2 Balíme balíček

Náš balíček zatím obsahuje jediný modul, Hodiny, a v něm funkci DllEntryPoint().[58] Připojíme k němu tedy již hotový modul s komponentou. Stiskneme tlačítko Add a v dialogovém okně, které se objeví, zadáme na kartě Add Unit v poli Unit file name jméno zdrojového souboru Hodinky.cpp (včetně cesty; můžeme použít tlačítko Browse).

[58] Připomeňme si, že balíček není nic jiného než zvláštní druh dynamické knihovny.

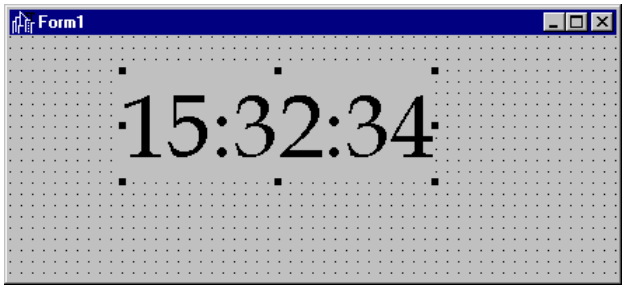
Nyní nezbývá, než balíček přeložit; použijeme tlačítko Compile ve správci balíčků. Pokud najde překladač nějaké chyby, ukáže nám je podobně jako při překladu „obyčejného“ programu. Naše komponenta by ale měla být už odladěná.

Pak balíček a s ním i komponentu instalujeme. K tomu nám poslouží tlačítko Install ve „správci balíčků“. Pokud je vše v pořádku, objeví se dialogové okno s informací, že se instalace podařila, a v hlavním okně přibude na paletě nová stránka jménem Pokus obsahující naši komponentu (obr. 10.3).



Obr.10.3 Už je tam…

Nyní svou komponentu vyzkoušíme. Uzavřeme aktuální projekt, vytvoříme nový a do vizuálního návrhu programu vložíme komponentu Hodiny. Nastavíme-li nyní její vlastnost Enabled na true, hodiny se rozeběhnou a poběží už v době návrhu. Přitom naše komponenta má všechny vlastnosti komponenty Label — můžeme měnit písmo, umístění, velikost, zarovnání atd. Zdědila ovšem i ikonu, tedy bitovou mapu, která ji reprezentuje na paletě.



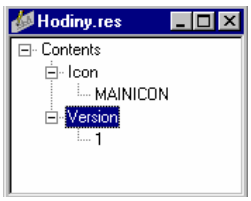
Obr. 10.4 Hodiny v komponentě ukazují přesný čas i v době návrhu

Poznámka: C++ Builder 3

Postup v předchozí verzi C++ Builderu se poněkud liší. Podobně jako ve verzi 4 vytvoříme nový balíček jménem Hodiny. Výsledkem bude projekt Hodiny.bpk a zdrojový text balíčku s funkcí DllEntryPoint(), podobně jako v předchozím případě. Pak příkazem Project | Add to Project vyvoláme dialogové okno Add. Na kartě Add unit v poli Unit file name zadáme jméno modulu, tj. Hodinky.cpp, a stiskneme OK. Pak spustíme překlad příkazem Project | Make z hlavní nabídky.

Ikona komponenty

Nakonec změníme bitovou mapu, která představuje naší komponentu na paletě i ve vizuálním návrhu programu. Příkazem Tools | Image Editor spustíme editor obrázků. V něm příkazem File | Open otevřeme soubor Hodiny.res s přeloženými prostředky naší komponenty; objeví se okénko, které je schématicky zobrazí (obr. 10.5).

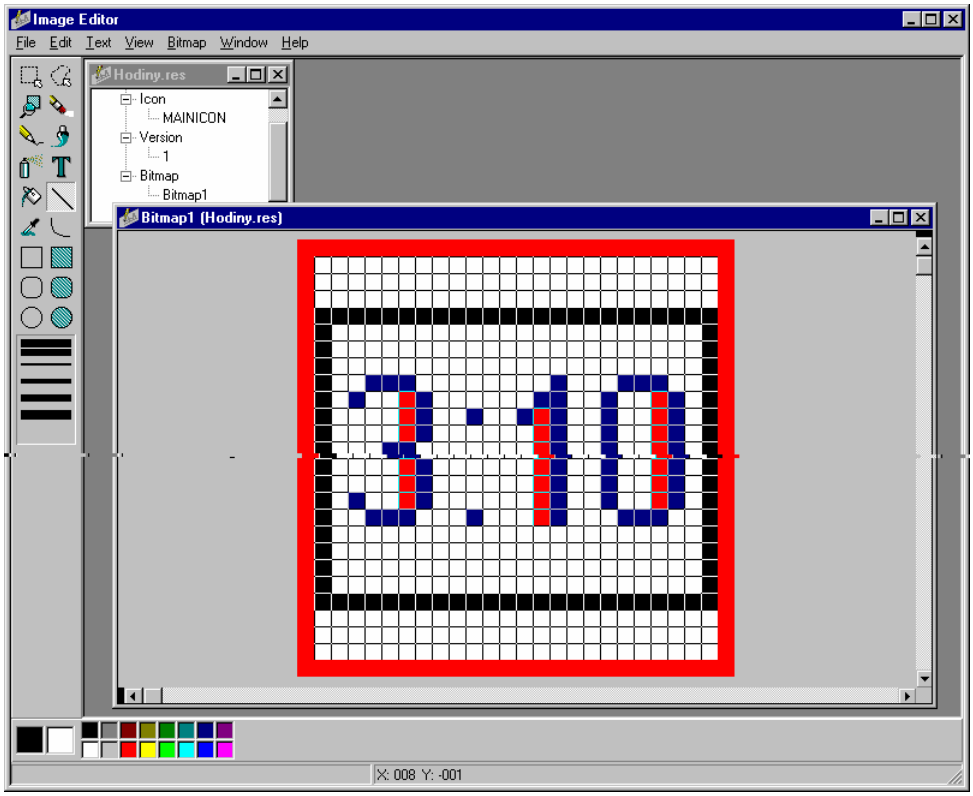


Obr. 10.5 Struktura prostředků v projektu komponenty

Zatím zde najdeme pouze ikonu a informaci o verzi. Obrázek, který reprezentuje komponentu na paletě, ale není ikona, nýbrž bitová mapa o rozměrech 24 × 24 pixelů. Použijeme tedy příkaz Resource | New | Bitmap a v dialogovém okně, které se objeví, zadáme příslušné rozměry. Bitová mapa přibude pod názvem Bitmap1 mezi prostředky našeho projektu. Dvojklikem na její ikonu ji otevřeme a v editoru nakreslíme vhodný obrázek, např. jako na obr. 10.6.

Pak tento obrázek přejmenujeme, např. tak, že kliknutím vybereme jméno Bitmap1 v okénku se schématem prostředků, v nabídce zadáme příkaz Resource | Rename a pak jméno prostě přepíšeme. Pozor: bitová mapa reprezentující komponentu se musí jmenovat stejně jako typ komponenty, ovšem psaný velkými písmeny; v našem případě tedy THODINKY.

Příkazem File | Open Project... znovu otevřeme projekt Hodiny.bpk; objeví se opět správce balíčků. Když nyní spustíme tlačítkem Compile překlad, upozorní nás C++ Builder nejprve, že se chystá znovu překládat balíček, a vyžádá si souhlas. Po novém překladu a instalaci bude mít komponenta objeví již novou ikonu.



Obr. 10.6 Kreslíme obrázek komponenty

Poznámka

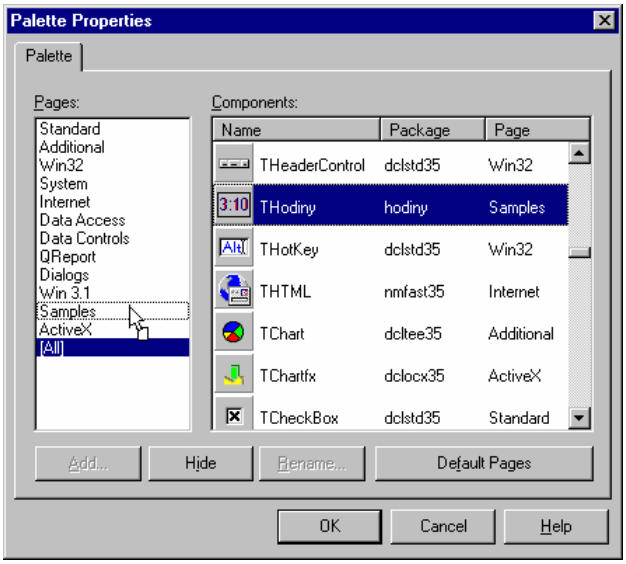
V C++ Builderu 3 uložíme bitovou mapu do souboru s příponou .DCR (např. obrazky.dcr) a do programu jej zařadíme pomocí makra USERES, které použijeme v souboru obsahujícím funkci DllEntryPoint():

```
USERES("obrazky.dcr");
```

Instalace cizí komponenty

Po vytvoření balíčku dostaneme soubor Hodiny.bpl, který obsahuje vlastní komponentu, soubor Hodiny.bpi, který obsahuje importní knihovnu pro balíček, a soubor Hodiny.lib, který obsahuje naši komponentu v podobě statické knihovny. K těmto souborům musíme dodat ještě hlavičkový soubor Hodiny.h s popisem třídy komponenty.

Při instalaci cizí komponenty umístíme hlavičkový soubor do adresáře s hlavičkovými soubory (standardně INCLUDE\VCL), soubory .bpi a .lib do adresáře se statickými knihovnami a soubory (standardně LIB) a soubor .BPL do adresáře s balíčky (např. BIN). Pak balíček instalujeme: Příkazem Component | Install Package vyvoláme dialogové okno Project Options se stránkou Packages, stiskneme tlačítko Add a v dialogovém okně zadáme jméno souboru s balíčkem. Pak Příkazem Component | Configure Palette vyvoláme okno Palette Properties, v poli Pages označíme All a v poli Components vyhledáme instalovanou komponenty. Myší ji pak přetáhneme na zvolenou stránku palety (obr. 10.7).



Obr. 10.7 Umístíme komponentu na zvolenou paletu

11. C++ Builder a programovací jazyk C++

Podívejme se nyní blíže na implementaci programovacích jazyků C a C++ v C++ Builderu 4.

11.1 Obecná charakteristika

C++ Builder 4 je založen na programovacím jazyku C++ v duchu standardu ANSI/ISO z roku 1997. Záměrně říkám „v duchu“, neboť není s tímto standardem zcela shodný; to nelze v současné době tvrdit o žádném z běžně používaných překladačů, neboť všechny se standardu teprve přibližují.

Existují tedy vlastnosti jazyka C++, které jsou popsány ve standardu, ale které tento překladač ještě neimplementuje. Vedle toho obsahuje C++ Builder některá rozšíření vynucená požadavkem kompatibility s Delphi a prostředím Win32.

Implementace programovacího jazyka C odpovídá standardu ANSI/ISO 9899-1990 s několika rozšířeními.

11.2 Jazyk C

Z rozšíření, se kterými se setkáme v jazyce C, jsou nejspíš nejdůležitější strukturované výjimky (konstrukce __try/__except a __try/__finally). Toto rozšíření se původně objevilo v microsoftských překladačích pro Win32; implementace v C++ Builderu neobsahuje klíčové slovo __leave.[59]

Vložené funkce

Mezi drobná, ale příjemná rozšíření patří klíčové slovo __inline, které umožňuje deklarovat vložené funkce i v jazyce C. Má naprosto stejný význam jako klíčové slovo inline, které je ale k disposici pouze v jazyce C++.

Celočíselné typy

S celočíselnými typy se zadávanou velikostí, __int8, __int16, __int32 a __int64, jsme se setkali už v předchozích verzích C++ Builderu. Tyto typy lze používat i s modifikátorem unsigned, např. unsigned __int64. Číselná přípona určuje velikost proměnných tohoto typu v bitech. Typ __int8 je ekvivalentní typu signed char, typ unsigned __int8 je ekvivalentní typu unsigned char; typ __int16 je ekvivalentní typu short a typ __int32 je ekvivalentní typu int. Typ __int64 neodpovídá žádnému ze standardních typů a představuje rozsah hodnot od -9223372036854775808 do 9223372036854775807, typ unsigned __int64 má rozsah od 0 do 18446744073709551615.

Pro zadávání konstant těchto typů jsou k dispozici přípony i8, i16, i32, i64, ui8, ui16 a ui64, např.

```
unsigned __int64 i = 0ui64;
```

Pozor, v případě bezznaménkových hodnot nelze na rozdíl od standardních přípon zaměnit pořadí u a i*N*, takže nelze psát např.

```
unsigned __int64 i = 0i64u; // NELZE
```

Při tisku hodnot typu __int64 pomocí funkce printf() nebo dalších funkcí ze standardní knihovny jazyka C použijeme modifikátor velikosti L, např.

```
unsigned __int64 i64 = 123456789123456789ui64;
printf("%Lu", i64);
```

Stejný modifikátor můžeme použít i při čtení těchto hodnot pomocí funkce scanf():

```
scanf("%Lu", &i64);
```

[59] Podrobný výklad o strukturovaných výjimkách najdete v češtině např. v [3] nebo v [9].

11.3 Jazyk C++

Implementace jazyka C++ se velmi blíží standardu tohoto jazyka. Přesto najdeme v C++ Builderu několik vlastností, které standard předepisuje, ale které ještě nejsou implementovány, a také některá rozšíření.

Vztah ke standardu

Zde stojí za zmínku, že C++ Builder dosud nedovoluje definovat operátor delete s „umístěním“, tj. s dodatečnými parametry, které by umožnily volat verzi operátoru delete odpovídající použitému operátoru new při výjimce v konstruktoru dynamické instance. Jde ovšem o konstrukci, která nepatří k běžně používaným.

Na druhé straně současná verze C++ Builderu obsahuje implementaci šablon odpovídající standardu, tj. včetně možnosti deklarovat šablonu jako složku třídy nebo šablony třídy, včetně možnosti specifikovat šablonové parametry šablon a včetně klíčového slova export.

Tyto konstrukce se ovšem uplatňují především ve standardní šablonové knihovně (STL, standard template library), a vůbec při tvorbě knihoven, a proto je zde nebudeme podrobněji rozebírat. Případné zájemce odkazuji na [3] a [9].

Rozšíření

Součástí implementace jsou i některá rozšíření, tedy konstrukce, které standard jazyka nezná. I když většinu z nich při běžném programování v C++ Builderu nepoužijeme, je třeba o nich alespoň rámcově vědět.

Nová klíčová slova

V Borland C++ Builderu 4 se setkáváme s několika novými klíčovými slovy. Ve stručnosti se zde podíváme na jejich význam; vezmeme je přitom v abecedním pořadí.

__automated

Používá se jako specifikátor přístupových práv v deklaraci třídy. Podobně jako public znamená, že za ním následují složky, které jsou dostupné všem částem programu. Pro tyto složky vygeneruje překladač informace nezbytné pro OLE automation. Vzhledem k tomu, že toto téma značně přesahuje rámec naší knihy, nebudeme ho zde rozebírat.

__classid

Toto klíčové slovo představuje operátor, jehož operandem je jméno třídy. Tento operátor vrátí ukazatel na tabulku virtuálních metod. Setkáme se s ním při vytváření formulářů v hlavním programu a při registraci komponent. Jinak není důvod ho používat.

__closure

Anglické slovo *closure* si dovolím poněkud svévolně přeložit výrazem *klauzura*. Klauzura je zvláštní typ ukazatelů na metody, který se skládá ze dvou částí — obsahuje adresu metody (tedy jejího kódu) a zároveň adresu instance, pro který se má tato metoda volat.

Tento druh ukazatelů se používá především při definici událostí pro komponenty. Každá z událostí je klauzura. Podíváme-li se do nápovědy, zjistíme, že např. událost OnClick je vlastnost typu TNotifyEvent

```
__property Classes::TNotifyEvent OnClick =
                        {read=FOnClick, write=FOnClick, stored=IsOnClickStored};
```

a typ je TNotifyEvent definován jako

```
typedef void __fastcall (__closure *TNotifyEvent)(System::TObject* Sender);
```

Jestliže definujeme handler pro tuto událost, znamená to, že se do vlastnosti OnClick (tedy do proměnné FOnClick) uložíme ukazatel na instanci a ukazatel na jeho metodu. Tato metoda se bude volat při příchodu události. Díky tomu můžeme měnit handlery pro události za běhu programu, můžeme používat jeden handler pro více různých událostí pro stejné i různé komponenty (pokud mají jejich handlery stejný počet a typ parametrů) apod.

Poznamenejme, že události by šlo implementovat, aniž bychom museli klauzury zavádět; jsou však nezbytné pro kompatibilitu s Delphi.

__declspec(specifikace)

S klíčovým slovem __declspec se v poslední době setkáváme v řadě překladačů pro Win32. Slouží k zadávání nestandardních modifikátorů. Syntakticky se chová jako deklarační specifikátor a v závorkách za ním následuje zadávaný nestandardní modifikátor. V C++ Builderu se můžeme setkat mj. s následujícími konstrukcemi:

__declspec(dllexport)	funkce nebo třída exportovaná z dynamické knihovny,
__declspec(dynamic)	slouží ke specifikaci tzv. dynamických funkcí (viz dále),
__declspec(package)	říká, že kód definující třídu může být přeložen jako součást balíčku,
__declspec(delphireturn)	pro kompatibilitu s Pascalem,
__declspec(delphiclass)	pro kompatibilitu s Pascalem,
__declspec(pascalimplementation)	pro kompatibilitu s Pascalem,
__declspec(hidebase)	pro kompatibilitu s Pascalem.

Poslední čtyři modifikátory najdeme především v hlavičkových souborech .hpp vytvořených překladačem Pascalu pro použití v C++.

__property

Definuje vlastnost komponenty. Připomeňme si, že *vlastnost* v této souvislosti složku instance objektového typu, jejíž změna se okamžitě promítne do skutečných vlastností této instance. Je-li např. okno ukazatel na instanci třídy TForm, pak přiřazení

```
okno -> Visible = true;
```

způsobí, že instance *okno začne být viditelná — zobrazí se na obrazovce.

Budeme o nich podrobněji hovořit dále.

__published

Používá se jako specifikátor přístupových práv v deklaraci třídy, a to ve dvou významech. Podobně jako public znamená, že za ním následují složky, které jsou dostupné všem částem programu.

Použijeme-li tento specifikátor pro vlastnost v deklaraci komponenty, vygeneruje pro ni překladač informace nezbytné pro zobrazení této vlastnosti v inspektoru objektů.

V deklaraci třídy okna nebo datového modulu označuje specifikace __published část, kterou vytvořil C++ Builder za základě vizuálního návrhu a do které by programátor jinak než prostřednictvím vizuálního návrhu neměl zasahovat.

Vlastnosti

Deklarace vlastnosti komponenty má tvar

__property *typ identifikátor* =
{read = *jméno*, write = *jméno*, stored = *výraz*, default = *hodnota*};

V tomto popisu jsou neproporcionálním písmem zapsány terminální symboly, tj. části, které musíme do programu opsat, a *kurzivou* neterminální symboly, tj. části, za které musíme „něco dosadit“.

Deklarace vlastnosti se tedy podobá deklaraci datové složky, až na to, že za ní následují jakési podivné specifikace. Poznamenejme, že kterákoli ze specifikací ve složených závorkách může chybět.

Čtení a zápis hodnoty vlastnosti

Za specifikací read může následovat buď jméno datové složky, ve které je hodnota vlastnosti uložena, nebo jméno metody, která se bude volat vždy při použití hodnoty této vlastnosti.

Za specifikací write může následovat buď jméno datové složky, je které je hodnota vlastnosti uložena, nebo jméno metody, která se bude volat vždy při přiřazení nové hodnoty této vlastnosti.[60]

Tato proměnná a metody je zpravidla deklarována jako chráněná (protected).

Implicitní hodnota vlastnosti

Za specifikací default následuje výraz udávající implicitní hodnotu dané vlastnosti, tj. hodnotu, kterou bude mít, pokud ji v inspektoru objektů nebo za běhu programu přiřazením nezměníme. Ovšem, pozor — to neznamená, že se překladač o přiřazení této hodnoty při vytvoření instance komponenty automaticky postará, to musíme zařídit sami v konstruktoru této komponenty. Specifikace implicitní hodnoty umožňuje C++ Builderu rozhodnout, zda se bude hodnota vlastnosti ukládat do souboru .dfm s popisem formuláře. (Připomeňme si, že do tohoto souboru se hodnoty vlastností ukládají pouze v případě, že se liší od implicitní hodnoty.)

Ukládání vlastnosti

Za specifikací stored následuje logický výraz, který určuje, jak se bude hodnota vlastnosti ukládat do souboru .dfm při uzavření projektu. Bude-li za slovem stored zapsána hodnota true, bude se hodnota vlastnosti ukládat, bude-li tam false, ukládat se nebude. Tato hodnota se později použije při opakovaném otevření projektu nebo při zobrazení formuláře za běhu programu.

Implicitně se hodnoty vlastností definovaných v sekci __published ukládají automaticky, hodnoty vlastností definovaných v ostatních sekcích deklarace třídy komponenty (public atd.) nikoli. To znamená, že pomocí této specifikace můžeme zabránit ukládání hodnot vlastností ze sekce __published a předepsat ukládání hodnot vlastností z ostatních sekcí.

Výjimky

Výjimky, které vzniknou v komponentách, musíme zachycovat odkazem nebo výpustkou.

Strukturované výjimky

Strukturované výjimky nejsou specialitou C++ Builderu, a proto zde o nich podrobněji nehovoříme. V C++ Builderu 4 ovšem nacházíme jednu významnou novinku: Bloky s koncovkou, tedy konstrukce __try/__finally, můžeme používat i v C++.

Datové typy

V C++ Builderu se setkáváme s několika novými datovými typy.

Celočíselné typy

O typech __int8, __int16, __int32 a __int64, jsme hovořili již v souvislosti s jazykem C. Zde pouze zdůrazníme, že typ __int8 je jen jiným jménem pro typ char, __int32 je jen jiným jménem pro int atd. To znamená, že pokud napíšeme

```
__int8 c = 65;
cout << c << endl;
```

vypíše program 'A'. Dále to znamená, že nemůžeme vedle sebe definovat např. přetížené funkce

```
void f(__int16 x) { /* ... */ }
void f(short y)  { /* ... */ }       // Nelze, typ parametrù se neliší
```

[60] V C++ Builderu i v Delphi se dodržuje konvence, že jméno proměnné, ve které je hodnota vlastnosti uložena, se skládá ze jména vlastnosti, před kterým je navíc písmeno F. Metoda pro získání hodnoty vlastnosti (uvedená u specifikace read) se skládá ze jména vlastnosti, před kterým je navíc slovo Get, a metoda pro uložení hodnoty vlastnosti (uvedená u specifikace write) se skládá ze jména vlastnosti, před kterým je navíc slovo Set. Doporučuje se tyto konvence dodržovat.

Množiny

V některých případech potřebujeme množinu příznaků. Např. písmo (font) může být tučné, podtržené nebo kurziva; ale může to také být tučná podtržená kurziva nebo nemusí mít žádnou z těchto vlastností. Typ písma je tedy popsán množinou příznaků.

Podobně chceme-li vědět, které přeřaďovače byly stisknuty při kliknutí myší, potřebujeme množinu, neboť nemusel být stisknut žádný a mohla být stisknuta třeba kombinace CTRL+ALT.

Množiny jsou v C++ Builderu navrženy jako analogie typu množina v Pascalu, tedy jako proměnné obsahující bitové příznaky určující, který z možných prvků v množině je a který tam není. Prvky množiny mohou být např. znaky nebo hodnoty výčtových typů. Jsou implementovány pomocí šablony

```
template<class T, unsigned char minEl, unsigned char maxEl>
class __declspec(delphireturn) Set;
```

kde T je typ ukládaných hodnot a minEl, resp. maxEl je hodnota nejmenšího, resp. největšího možného prvku. V C++ Builderu je definováno několik instancí této šablony, např.

```
enum TFontStyle { fsBold, fsItalic, fsUnderline, fsStrikeOut };
typedef Set<TFontStyle, fsBold, fsStrikeOut> TFontStyles;
```

Jiným příkladem je typ TShiftState vyjadřující množinu stisknutých přeřaďovačů.

Pro přidávání prvků do množiny slouží přetížený operátor <<, pro vyjímání prvků z množiny operátor >>. Chceme-li zjistit, zda je nějaká hodnota prvkem množiny, použijeme metodu Contains(). Pro běžné operace s množinami lze použít operátory + (sjednocení), * (průnik), - (rozdíl), = (přiřazení), == (rovnost), != (nerovná se).

Objektové typy

Objektové typy definované v C++ Builderu, které nijak nesouvisí s VCL, se chovají po všech stránkách tak, jak předepisuje standard jazyka C++, a proto o nich není třeba hovořit. Problémy ovšem nastanou, podíváme-li se blíže na součásti knihovny VCL a na třídy od nich v C++ Builderu odvozené. Zde musíme mít na paměti několik omezení:

- ✧ Instance tříd, které jsou přímo nebo nepřímo odvozené od třídy TObject, musí být dynamické. To znamená, že nelze napsat

  ```
  TEdit E;        // NELZE
  ```

  ale musí být

  ```
  TEdit *ue = new Tedit;      // OK
  ```

- ✧ Pro tyto třídy nelze používat vícenásobnou dědičnost (a tedy ani virtuální dědičnost).
- ✧ Pro metody těchto tříd nelze definovat implicitní hodnoty parametrů.
- ✧ Instance nelze přiřazovat a nejsou pro ně definovány kopírovací konstruktory. To mj. znamená, že jako parametry funkcí je smíme předávat pouze odkazem nebo ukazatelem.

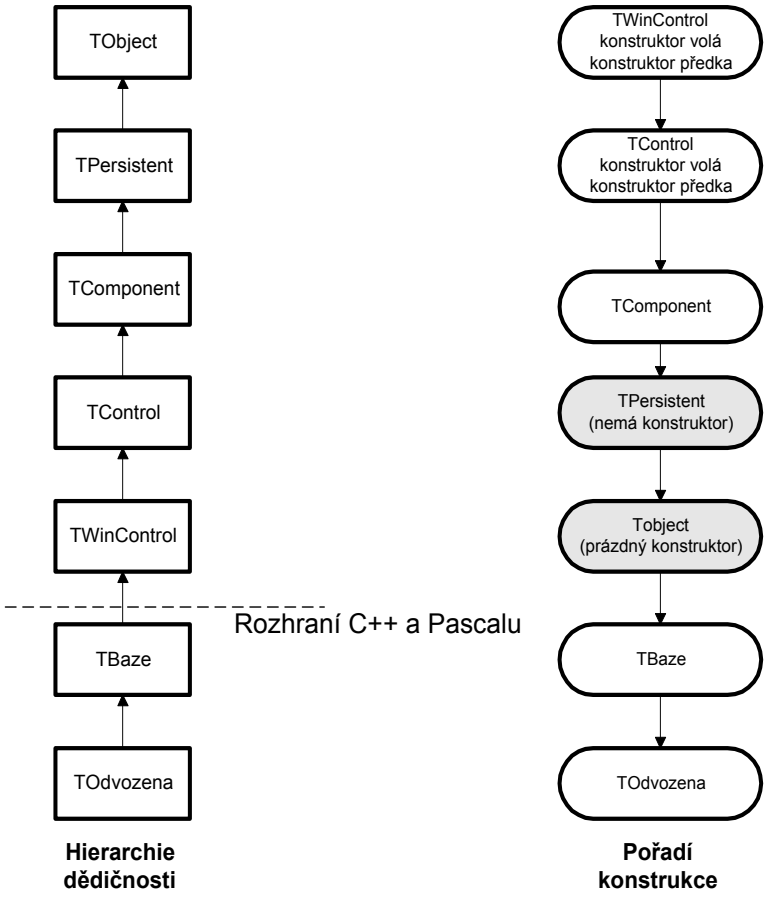
Vytváření instancí

Knihovna VCL byla napsána v Object Pascalu. Odtud plynou nejen výše uvedená omezení, ale i některé další problémy. Standard jazyka C++ předepisuje, že konstruktor odvozené třídy, potomka, vždy nejprve zavolá konstruktory svých bezprostředních předků. To znamená, že v době, kdy se začne provádět tělo konstruktoru potomka, jsou zděděné podobjekty již plně zkonstruovány (inicializovány).

Nic takového ale v Object Pascalu neplatí. V Pascalu může konstruktor potomka volat konstruktor předka, ale nemusí, a pokud ho volá, může ho volat kdykoli. (Není to automatické jako v C++, programátor musí zavolat konstruktor předka explicitně. Pokud se konstruktor předka nezavolá, musí se o inicializaci složek předka postarat potomek; paměť pro složky předka se samozřejmě vyhradí.)

Třídy odvozené v C++ Builderu od tříd z knihovny VCL se chovají jako ostatní objektové typy definované v C++, tj. jejich konstruktor nejprve volá konstruktor předka a teprve pak proběhne jeho vlastní tělo. Ovšem konstrukce vzdálenějších předků — těch, kteří jsou definováni v knihovně VCL, a tedy napsáni v Pascalu — probíhá podle pravidel Pascalu.

Podívejme se na příklad. Definujeme v C++ Builderu třídu Baze, která bude potomkem typu TWinControl z VCL, a od ní pak odvodíme třídu TOdvozena.



Obr. 11.1 Pořadí konstrukce objektů odvozených od součástí knihovny VCL

```
class TBaze: public TWinControl
{
  public:
   TBaze();
};

class TOdvozena: public TBaze
{
  public:
   TOdvozena();
};

TOdvozena *uo = new TOdvozena;
```

Dědickou hierarchii, tedy posloupnost předků až po třídu TObject, která je základem knihovny VCL, ukazuje levá část obr. 11.1. V pravé části pak vidíme posloupnost volání konstruktorů.

Poznamenejme, že kdyby se jednalo o hierarchii plně definovanou v C++, volal by se jako první konstruktor třídy, Tobject (tj. jako první by se konstruoval zděděný podobjekt třídy TObject), pak by následoval konstruktor, třídy TPersistent, … , pak konstruktor třídy TWinControl a nakonec konstruktor třídy TBaze a TOdvozena.

Virtuální metody

Vedle virtuálních metod tak, jak je známe z C++, se v knihovně používají i tzv. dynamické metody. Deklarují se pomocí specifikátoru __declspec(dynamic).

Jejich smysl, a tedy i jejich fungování z hlediska chodu programu, je stejný jako v případě virtuálních metod. Jediné, čím se liší, je způsob překladu volání. Použití dynamických metod vede v případě knihoven, jako je VCL, k menšímu, ale také k pomalejšímu výslednému programu než v případě „obyčejných“ virtuálních metod.

Poznamenejme, že dynamické metody lze používat pouze pro třídy odvozené od některé ze tříd z knihovny VCL.

Programátorům zvyklým na C++ může připadat podivné, že konstruktory komponent mohou (a v některých případech i musí) být virtuální, ale je to tak. Ani tuto konstrukci nelze použít pro třídy, které nejsou potomky typu TObject.

12. Borland C++ Builder a jeho prostředí

Integrované vývojové prostředí C++ Builderu 4 (IDE — Integrated Development Environment) je pokračováním vývojových prostředí předchozích verzí, obsahuje ale řadu novinek (většinou příjemných, ale záleží i na kvalitě počítače, na kterém je budete používat). V této kapitole budeme hovořit o zacházení s ním.

12.1 Projekt

Před začátkem vlastní práce musíme určit druh projektu. K tomu slouží příkaz File | New, který vyvolá zásobník objektů (okno Object Repository, obr. 2.1). Aplikace založená na formulářích (oknech) a knihovně VCL se skrývá na stránce New pod označením Application; chceme-li konzolovou aplikaci nebo okenní aplikaci vytvořenou ve Windows API, zvolíme Console Wizard.

Vytváření některých specifických druhů aplikací mám může usnadnit řada šamanů neboli wizardů. Můžeme tak získat nejen základ aplikace využívající knihoven MFC nebo OWL (odpovídající šamani jsou v zásobníku objektů na straně New), ale i základ běžné aplikace založené na VCL (na kartě Projects), databázové aplikace nebo aplikace pro WWW (na kartě Business), řídicího prvku ActiveX (na kartě ActiveX) atd.

Zacházení s projektem

Podívejme se ve stručnosti na základní operace s projekty v C++ Builderu.

Správce projektů

Základní orientaci v projektu umožňuje správce projektů (Project Manager, obr. 1.3), kterého vyvoláme příkazem View | Project Manager nebo klávesovou zkratkou CTRL+ALT+F11. Toto okno zobrazuje strukturu skupiny projektů a jednotlivých projektů podle modulů. (Tedy ukazuje hlavní program a jednotlivé moduly, např. okno, a případně soubory a formuláře, které modul tvoří, např. okno.cpp a HlavniOkno. Dvojklikem na modul, případně na soubor přejdeme do zdrojového textu nebo do vizuálního návrhu okna.

Časté operace s projekty

Otevření existujícího projektu: Příkazem File | Open Project... nebo klávesovou zkratkou CTRL+F11 vyvoláme dialogové okno Open Project a v něm zvolíme projektový soubor, který chceme otevřít.

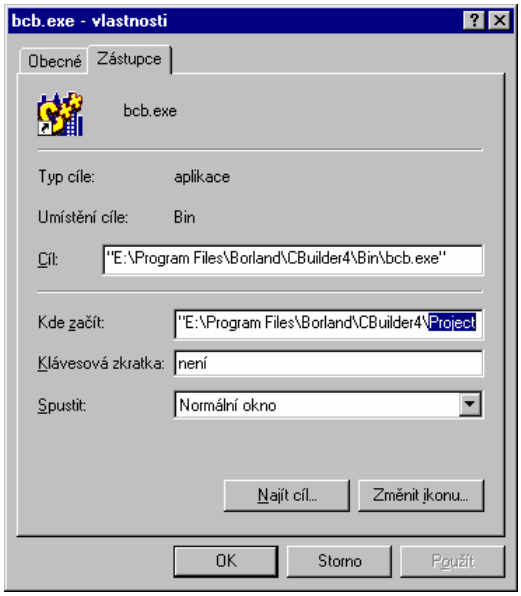
Můžeme také zkusit příkaz File | Reopen. Prostředí nám nabídne podřízenou nabídku se seznamem projektů a souborů, se kterými jsme v poslední době pracovali.

Uložení projektu: Projektový soubor se uloží při příkazu File | Save All nebo při stisknutí tlačítka Save All na nástrojovém panelu. Při prvním uložení si prostředí vyžádá jméno projektu.

Uložení projektu pod jiným jménem: Příkazem File | Save Project As... vyvoláme dialogové okno, které nám umožní uložit projekt pod jiným jménem. Ovšem pozor, uloží se pouze soubor .bpr s popisem projektu, soubor .res s přeloženými prostředky projektu a soubor .cpp s hlavním programem. Odkazy na další moduly zůstanou beze změny.

Uzavření projektu: Příkazem File | Close All.

Implicitně nabízený adresář projektu: Při prvním ukládání projektu nám prostředí nabídne adresář, do kterého náš projekt uloží. Měl by to být adresář Borland \ CBuilder4 \ Projects, ale v některých instalacích to bude adresář BIN s instalací Builderu, což je krajně nevhodné. Implicitně nabízený adresář změníme takto: Klikneme pravým tlačítkem myši na ikonu programu a z příruční nabídky vybereme Vlastnosti. Objeví se dialogové okno Vlastnosti, ve kterém přejdeme na kartu Zástupce. Požadovaný adresář zadáme v poli Kde začít.



Obr. 12.1 Měníme implicitně nabízený adresář projektů

Připojení souboru k projektu: Chceme-li připojit k projektu nový soubor, použijeme příkazu Project | Add to Project (SHIFT+ F11). Jméno souboru zadáme v dialogovém okně, které se objeví.

Součástí projektu v C++ Builderu mohou být zdrojové programy napsané v jazyce C++, C, Pascalu, Turbo Assembleru. Dále můžeme k projektu připojit soubor s prostředky (.RC, .RES), relativní modul (.OBJ), knihovní modul (.LIB), definiční soubor modulu (.DEF), typovou knihovnu (.TLB) nebo soubor s popisem rozhraní (.IDL). Poslední dvě možnosti se týkají programování distribuovaných aplikací (COM, CORBA).

Odstranění modulu z projektu: Chceme-li z projektu odstranit modul, označíme ho ve správci projektů a stiskneme tlačítko Remove v tomto okně. (Můžeme také použít klávesu DEL.) Jinou možnost představuje příkaz Project | Remove from Project, který vyvolá dialogové okno obsahující jednotlivé moduly.

Skupina projektů

Skupiny lze sdružovat do projektů. Nový, resp. existující projekt připojíme do skupiny příkazem Project | Add New Project, resp. Project | Add Existing Project.

Překlad, sestavování a spouštění

Překlad jednotlivého modulu: Příkaz Project | Compile Unit.

Překlad a sestavení celého projektu, resp. skupiny projektů: Příkazy Project | Make Project (CTRL+F9), resp. Project | Build Project. Překlad a sestavení celé skupiny projektů spustíme příkazy Project | Make All Projects, resp. Project | Build All Projects. Příkaz Make přeloží jen soubory, u kterých to je nezbytné (které se od předchozího překladu změnily), příkaz Build přeloží vždy všechny soubory. Poznamenejme, že při některých operacích — např. při přenosu projektu z předchozí verze nebo při změně ikony — je nezbytné použít příkaz Build.

Spuštění přeloženého programu v prostředí: Příkazem Run | Run nebo klávesovou zkratkou F9. O zadávání parametrů příkazové řádky programu budeme hovořit v oddílu věnovaném ladění.

Překladač, linker, preprocesor

Překladač BCC32 verze 5.4 je spolu s dalšími soubory instalován v podadresáři BIN domovského adresáře C++ Builderu. Umí překládat zdrojové soubory v jazycích C a C++ (rozlišuje je podle přípony) a lze jej spustit i z příkazové řádky. Spolu s ním zde najdeme i překladač Pascalu DCC32 verze 12.5.

Překladač BCC32 umí také překládat C nebo C++ do asebmleru. K tomu slouží volba -S v příkazové řádce, např.

```
bcc32 -S X.CPP
```

K sestavování se používá inkrementální linker ILINK32.

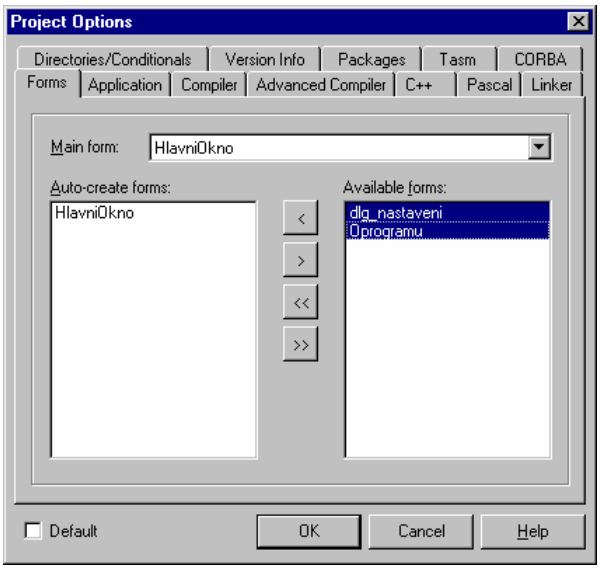
Preprocesing, tedy rozvoj maker, zpracování direktiv #include ap., je součástí překladu. Ovšem hledání chyb v makrech patří k méně příjemným stránkám programování v C/C++, a proto je součástí instalace i samostatný preprocesor CPP32. Lze jej spustit z příkazové řádky nebo instalovat do nabídky Tools. Vytvoří soubor se stejným jménem jako má zdrojový soubor a s příponou .I.

Nastavení projektu

Nastavení pro překlad a sestavování programu zadáváme v dialogovém okně, které vyvoláme příkazem Project | Options. Je na něm řada karet s volbami, o jejichž významu si zde krátce povíme. Ve spodní části okna je zaškrtávací pole Default. Zaškrtneme-li je, uloží se aktuální nastavení jako implicitní.

Forms

V poli Main form na této kartě určujeme hlavní okno aplikace, tj. okno, které se zobrazí po spuštění aplikace.

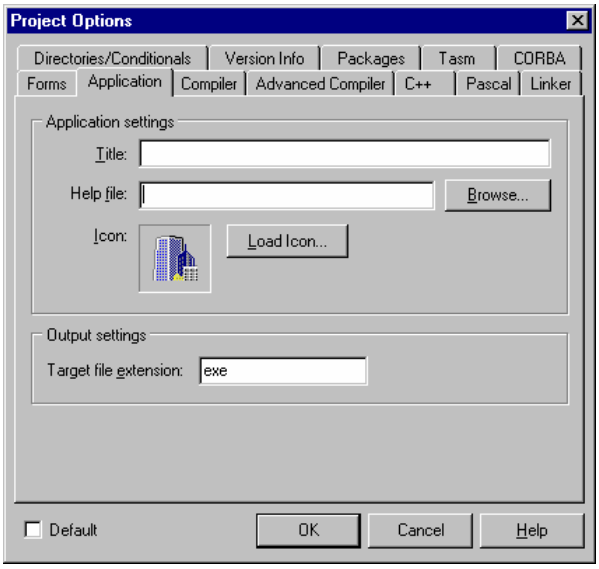


Obr. 12.2 Práce s okny

V polích Auto-create forms a Available forms najdeme seznam všech tříd oken aplikace. Okna uvedená v seznamu Auto-create se po spuštění aplikace vytvoří automaticky, vytvoření oken v seznamu Available musíme sami naprogramovat. Chceme-li okno přemístit z jednoho seznamu do druhého, vybereme ho myší a stiskneme tlačítko se znaménkem > nebo <. Tlačítka >> a << přemístí všechna okna v dané skupině. Okna lze také přemísťovat myší metodou táhni a pusť.

Application

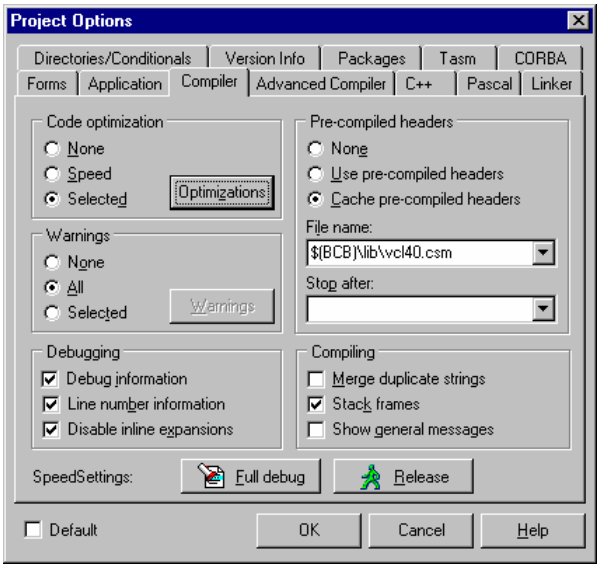
V poli Title určíme titulek aplikace, tj. text, který se objeví pod ikonou, resp. vedle ikony v pruhu úloh. V poli Help file předepíšeme soubor s nápovědou a v poli Icon se zobrazuje aktuální ikona aplikace. Můžeme ji změnit pomocí tlačítka Load Icon.... Pole Target file extension určuje příponu vytvořeného souboru (zpravidla .EXE nebo .DLL).



Obr. 12.3 Základní vlastnosti aplikace

Compiler

Na této kartě nastavujeme základní volby pro překladač (obr. 12.3). Základní nastavení můžeme určit pomocí tlačítek Full debug (ladění) a Release (ostrá verze). Jednotlivé slupiny přepínačů umožňují podrobnější nastavení.



Obr. 12.4 Nastavení překladače

Skupina Code Optimizations nabízí možnosti None (žádné optimalizace), Speed (optimalizace na rychlost) a Selected (vybrané). Pokud zvolíme třetí možnost, můžeme stisknout tlačítko Optimizations a vyvolat okno, ve kterém lze zaškrtnout jednu z následujících možností:

- Pentium scheduling — optimalizace pořadí instrukcí pro superskalární architekturu procesoru Pentium (tento procesor má,jak známo, dvě aritmeticko-logické jednotky, a proto mohou některé operace probíhat zároveň,
- Inline intrinsic functions — některé z běžných knihovních funkcí se budou překládat jako vložené (tj. jako funkce s modifikátorem inline),
- Induction variables — slouží k optimalizaci cyklů,
- Optimize common subexpressions — odstranění opakujících se společných podvýrazů v rámci jedné funkce.

Skupina Warnings umožňuje potlačit některá varování. Možnost None znamená, že se nebudou vypisovat žádná varování, možnost All znamená, že se budou vypisovat všechna varování, a zvolíme-li Selected, budeme si moci po stisknutí tlačítka Warnings vybrat.

Skupina Debugging obsahuje volby pro ladění. Zaškrtneme-li Debug information, bude výsledný soubor OBJ obsahovat ladicí informace, zaškrtneme-li Line number information, bude soubor OBJ obsahovat informace o číslech řádek, a zaškrtneme-li Disable inline expansions, bude překladač ignorovat klíčové slovo inline. (To je důležité při krokování funkcí.)

Ve skupině Precompiled headers můžeme určit způsob používání předkompilovaných hlavičkových souborů (záhlaví). Možnost None znamená, že je překladač nevytváří, možnost Use pre-compiled headers znamená, že překladač bude předkompilovaná záhlaví vytvářet a používat, a možnost Cache pre-compiled headers způsobí, že překladač bude předkompilovaná záhlaví kešovat. To může být užitečné, pokud chceme předkompilovat více souborů.

V poli File name a Stop after můžeme zadat jméno souboru s předkompilovanými záhlavími a říci, po zpracování kterého z předkompilovaných souborů skončit.

Ve skupině Compiling nejdeme volby, které se zřejmě jinam nehodily. Možnost Merge duplicate strings způsobí, že překladač bude slučovat shodné znakové řetězce. Zaškrtneme-li možnost Stack frames, bude překladač generovat standardní rámec zásobníku i pro funkce, které nepoužívají parametry ani lokální proměnné. Program sice bude o něco větší, ale výrazně to usnadní ladění. Poslední možnost, Show general messages, způsobí, že překladač bude „upovídanější“, bude vypisovat i jiné zprávy než jen chyby a varování.

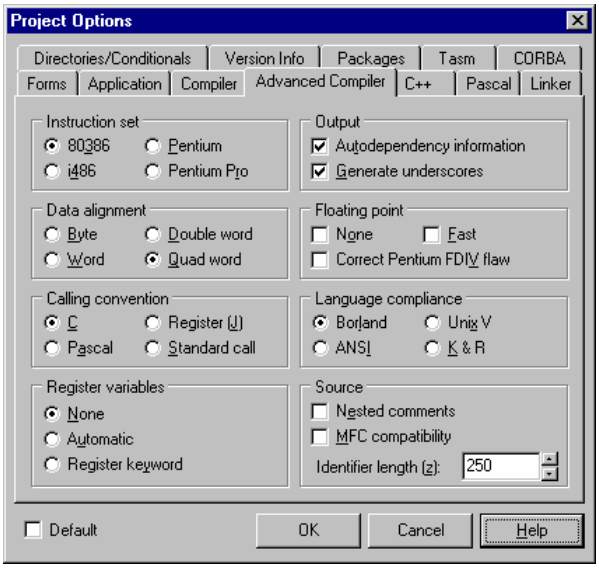
Advanced Compiler

Na této kartě (obr. 12.4) najdeme volby, které určují způsob překladu některých konstrukcí z jazyků C a C++.

Skupina přepínačů Instruction set určuje procesor, pro který bude přeložený program určen.

Skupina Data Alignment určuje zarovnání dat v paměti na hranici jednoho bajtu (Byte), slova (Word), 4 nebo 8 bajtů (Double word, Quad word). Zarovnání se týká nejen samostatných proměnných, ale i složek struktur a objektů. Takovéto zarovnání urychlí program, znamená ovšem větší spotřebu paměti.

Skupina Calling convention určuje implicitní volací konvenci, tj. konvenci, kterou překladač použije pro funkce, u kterých ji explicitně nespecifikujeme. Možnost C znamená standardní volací konvenci jazyka C (odpovídá jí klíčové slovo __cdecl), Pascal znamená konvenci používanou v jazyce Pascal (klíčové slovo __pascal), Register znamená registrovou volací konvenci (klíčové slovo __fastcall, je povinná pro handlery, konstruktory komponent, synchronizované metody ad.) a Standard call znamená volací konvenci používanou ve Win32 API (klíčové slovo __stdcall).



Obr. 12.5 Další volby překladače

Ve skupině Register variables určujeme způsob zacházení s registrovými proměnnými. První možnost, None, říká, že žádné proměnné nebudou ukládány do registrů; to se může hodit při ladění. Druhá možnost, Automatic, ponechává rozhodnutí o umístění proměnné zcela na úvaze překladače (překladač

bude klíčové slovo register ignorovat). Poslední možnost, Register keyword, přikazuje — samozřejmě pokud je to možné — ukládat do registrů proměnné, u kterých to programátor předepsal pomocí klíčového slova register.

Ve skupině Output najdeme dvě zaškrtávací pole. První z nich, Autodependency Information, způsobí automatickou kontrolu závislostí u každého cílového souboru, ke kterému je v seznamu projektu uveden odpovídající zdrojový soubor. Možnost Generate Underscores způsobí, že se před jména funkcí automaticky připojí znak podtržení „_“. (Týká se ale jen funkcí deklarovaných extern "C" nebo funkcí překládaných v jazyce C.)

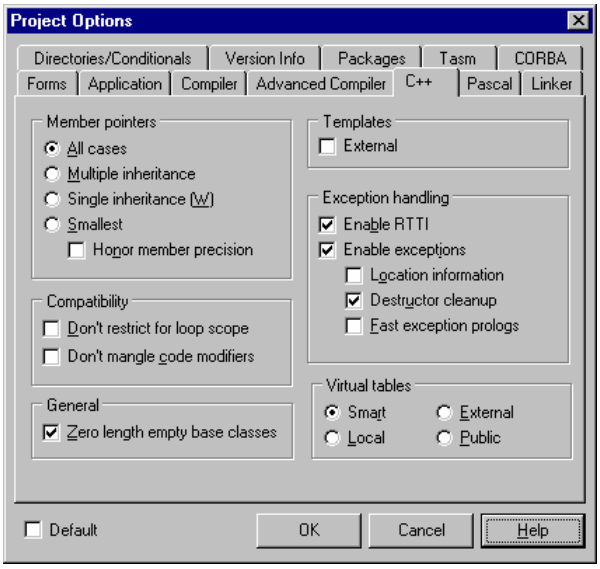
Ve skupině Floating Point najdeme volby týkající se práce s reálnými čísly. Možnost None znamená, že aritmetiku reálných čísel nebudeme vůbec používat a k programu se tedy nepřipojí odpovídající knihovny. Možnost Fast znamená, že operace s reálnými čísly budou optimalizovány bez ohledu na explicitní přetypování. (Překladač tedy bude ignorovat striktní pravidla pro konverze reálných čísel uvedená v normě ANSI.) Poslední možnost, Correct Pentium FDIV flaw, nabízí možnost softwarově korigovat chyby při dělení, které se vyskytovaly u prvních procesorů Pentium.

Přepínače ve skupině Language compliance určují, zda chceme v programu používat borlandská rozšíření jazyka C++ (možnost Borland), rozšíření obvyklá v prostředí Unixu V (možnost Unix V), zda má náš program přesně odpovídat standardu ANSI/ISO (možnost ANSI) nebo zda požadujeme program v jazyce C tak, jak jej popsali B. W. Kernighan a D. Ritchie ve známé knize [8] (možnost K & R).

Ve skupině Source najdeme přepínače, které určují zacházení se zdrojovým textem. Možnost Nested Comments povoluje nebo zakazuje vnořování komentářů. (Připomeňme si, že standardy jazyků C a C++ nedovolují vnořování komentářů začínajících znaky /* a končících */.) Přepínač MFC Compatibility povoluje nebo zakazuje rozšíření jazyka nezbytná pro překlad programů využívajících knihovny Microsoft Foundation Classes (MFC). Editační pole Identifier Length umožňuje předepsat omezení délky pro identifikátory. (Omezení délky identifikátoru může být užitečné, požadujeme-li kompatibilitu s některými unixovskými překladači.) Toto omezení se ovšem týká jen jazyka C, neboť standard jazyka C++ přímo stanoví, že délka identifikátoru není omezena. Zde můžeme zadat hodnoty v rozmezí 8 – 250.

C++

Na této kartě najdeme volby specifické pro jazyk C++. Ve skupině Member pointers určujeme způsob překladu třídních ukazatelů. Implicitně nastavená možnost All cases sice pokrývá všechny případy, ale znamená nejpomalejší program a největší spotřebu paměti. Pokud víme, že se v programu využívá vícenásobná, resp. pouze jednoduchá dědičnost, můžeme nastavit možnost Multiple Inheritance, resp. Single Inheritance.



Obr. 12.6 Volby specifické pro jazyk C++

Volba Smallest nutí překladač používat nejmenší možnou reprezentaci. Zaškrtnutím pole Honor member precision donutíme překladač používat vždy deklarovanou přesnost typu ukazatele. Má smysl při

explicitním přetypování ukazatele do potomka na ukazatel na předka v situaci, kdy ukazatel ve skutečnosti ukazuje na složku potomka.[61]

Volby ve skupině Compatibility ovládají kompatibilitu s překladači, které odpovídají starším verzím standardu. Zaškrtnutím Don't restrict for loop scope způsobíme, že překladač bude chápat zacházet s proměnnou deklarovanou v příkazu for jako s proměnnou deklarovanou na úrovni bloku, který tento příkaz obklopuje. (Podle současného standardu je proměnná deklarovaná v příkazu for lokální v tomto příkazu.) Druhá volba, Don't mangle code modifiers, se týká tzv. vnitřních jmen — tedy způsobu, jakým překladač upraví identifikátor funkce, aby byl jednoznačný. V současné verzi překladače se do vnitřního jména zakóduje i volací konvence, ve starších překladačích nikoliv. Zaškrtneme-li toto pole, nebude vnitřní jméno obsahovat kód volací konvence.

V skupině General najdeme jediné pole, Zero length empty base classes, v němž lze předepsat, aby měl zděděný podobjekt prázdné bázové třídy délku 0 bajtů. Tato volba může ovlivnit kompatibilitu s některými knihovnami. Také ve skupině Templates najdeme jediné pole, External, které určuje způsob generování instancí šablony.

Ve skupině Exception Handling najdeme 5 zaškrtávacích polí. První dvě, Enable RTTI a Enable Exceptions, určují, zda v programu budeme smět využívat dynamickou identifikaci typů (Run Time Type Identification, RTTI) a výjimky. Další tři jsou povoleny pouze v případě, že jsou povoleny výjimky. Zaškrtneme-li pole Location Information, budeme za běhu moci určit jméno souboru a číslo řádky, kde k výjimce došlo. Pokud zaškrtneme Destructor Cleanup, zavolají se po vzniku výjimky destruktory všech lokálních automatických instancí objektových typů v blocích mezi vznikem výjimky a handlerem, který ji zachytí. (To je standardní chování programu v C++ při výjimce.) Zaškrtneme-li Fast exception prologs, budou všechny funkce, které se o ošetřování výjimek starají, rozvinuty jako vložené (inline). Program bude rychlejší, ale také větší.

Ve skupině Virtual tables najdeme přepínače, které řídí zacházení s tabulkami virtuálních metod (VMT) a s funkcemi, které jsme deklarovali jako vložené, ale překladač je z nějakých důvodů nerozvinul. Možnost Smart vede k nejmenším a nejrychlejším programům, ale vytvořené soubory .obj jsou kompatibilní pouze s borlandským linkerem (a také asemblerem). Možnost způsobí, že každý modul bude mít svou lokální instanci VMT a své instance nerozvinutých vložených funkcí. Možnost External způsobí, že se budou generovat pouze odkazy na externí instance a možnost Public způsobí generování veřejně přístupných instancí.

Pascal

Na této kartě najdeme volby pro překladač Pascalu, podobné, jako v Delphi. Používání Pascalu v C++ Builderu ovšem není běžnou záležitostí, a proto se s nimi zde nebudeme zabývat. Jejich význam najdete v případě potřeby v nápovědě.

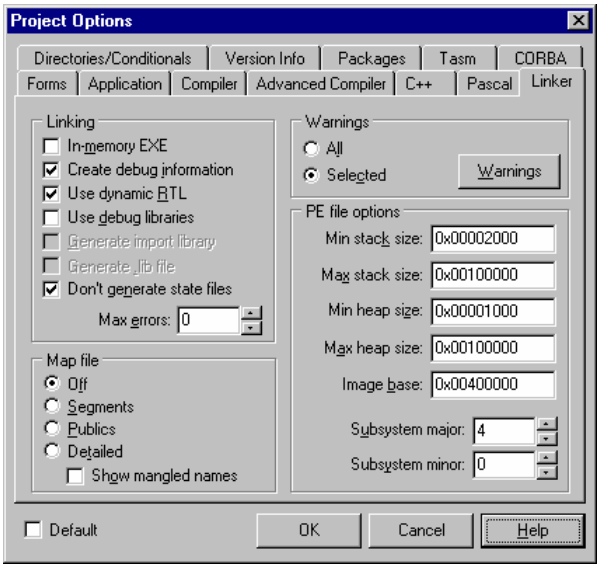
Linker

Nastavení na této kartě ovlivňují činnost sestavovacího programu. Ve skupině Linking najdeme volby, které určují základní chování linkeru. Zaškrtneme-li pole In-memory .EXE, způsobíme tím, že linker vytvoří spustitelný soubor pouze v paměti, nezapíše ho na disk. To může zrychlit práci při ladění programu. Možnost Create debug information přikazuje uložit do výsledného programu ladicí informace (samozřejmě pokud jsou v souborech .obj k disposici). Možnost Use dynamic RTL způsobí, že se standardní knihovna k programu připojí jako dynamická knihovna. Výsledkem je menší kód, ale zároveň s ním musíme distribuovat i soubor RTL.DLL. Zaškrtneme-li pole Use debug libraries, připojí se k programu ladicí verze knihovny VCL.

Pole Generate import library je k disposici pouze v případě projektů, jejichž výsledkem je dynamická knihovna. Zaškrtneme-li ho, vytvoří se zároveň i importní knihovna.

Pole Generate .lib file se vztahuje pouze k balíčkům (package). Pokud ho zaškrtneme, vytvoří se pro balíček importní knihovna .lib místo .bpi. To umožní připojit balíček k programu jako obyčejnou dynamickou knihovnu.

[61] Jde o zvláštní druh ukazatelů v C++, které mohou ukazovat na metody a nebo na složky instancí objektových typů. Pokud ukazují na složku instance, udávají vlastně relativní adresu složky vzhledem k počátku instance.



Obr. 12.7 Nastavení pro sestavovací program

Zaškrtneme-li pole Don't generate state files, zakážeme linkeru generovat stavové soubory a tím mu ve skutečnosti zakážeme přírůstkové (inkrementální) sestavování. Výsledkem bude, že všechna sestavování budou začínat od počátku a budou trvat stejně dlouho.

Číslo v poli Max errors určuje počet chyb, po nichž se linker zastaví.

Přepínače v poli Map file se vztahují k tzv. mapovému souboru vytvářenému linkerem. Možnost Off říká, že se tento soubor nemá vytvářet, možnosti Segments, resp. Publics přikazují, že aby tento soubor obsahoval pouze informace o segmentech programu, resp. o veřejně přístupných symbolech, a možnost Detailed říká, že chceme podrobný výpis. Zaškrtneme-li pole Show Mangled Names, přikážeme tím použít v mapovém souboru vnitřní jména.

Ve skupině Warnings určujeme, která varování má linker vypisovat. All znamená, že se budou vypisovat všechna varování, Selected, že jen vybraná. Stisknutím tlačítka Warnings vyvoláme dialogové okno se seznamem varování a v něm můžeme zaškrtnout ta, která se mají vypisovat.

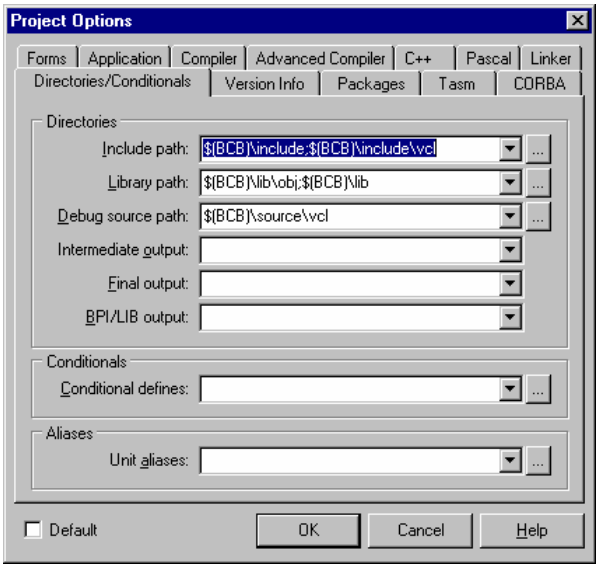
V poli PE File Options najdeme volby, které určují technické podrobnosti výsledného spustitelného souboru[62]. Čísla v polích Min stack size, Max stack size, Min heap size, Max heap size určují rozmezí velikosti zásobníku (stack) a haldy (heap).

Directories/Conditionals

Na této kartě určujeme adresáře, kde jsou uloženy knihovny, výsledné soubory atd. Jednotlivé cesty jsou odděleny středníkem a mohou obsahovat výraz $(BCB) odkazující na domovský adresář C++ Builderu. Vedle toho zde můžeme zadat makra, která mají být v průběhu překladu definována.

Ve skupině Directories najdeme skupinu vstupních řádek (polí) pro zadávání adresářů.

[62] PE (portable executable) je označení formátu spustitelného souboru pro Win32.



Obr. 12.8 Cesty a makra

V poli Include Path určujeme cestu k adresářům, ve kterých jsou hlavičkové soubory. Library Path předepisuje cestu ke knihovním souborům (.lib). Debug source path obsahuje cestu ke zdrojovému kódu pro ladění. Implicitně obsahuje cestu ke zdrojovému kódu knihovny VCL.

Další pole, Intermediate output, určuje adresář, do kterého se budou ukládat mezivýsledky překladu, tj. soubory .obj, .asm apod. Poznamenejme, že některé z „mezisouborů“, např. .res, se do tohoto adresáře neukládají.

Pole Final output určuje adresář, do kterého se uloží výsledné soubory (.exe, .dll, .bpl). Pole BPI/LIB output určuje adresář, do kterého se uloží importní knihovna, pokud vytváříme balíček (.bpl) nebo dynamickou knihovnu (.dll).

Ve skupině Conditionals najdeme pole Conditional defines, ve kterém můžeme zadat makra. Tato makra budou definována v průběhu celého překladu podobně, jako kdybychom je definovali direktivou #define. Identifikátory maker oddělujeme středníky; případnou hodnotu makra připojíme k identifikátoru znakem =, např. xxx;yyy=1

Ve skupině Aliases najdeme pole Unit aliases, které umožňuje zadat jiná jména pro pascalské moduly (unit). Toto pole slouží především pro zpětnou kompatibilitu, neboť v novějších verzích Delphi byly některé moduly přejmenovány. V tomto poli zadáváme položky ve tvaru <starý_modul>=<nový_modul> a oddělujeme je středníky.

Za každým z polí na této kartě je tlačítko se šipkou dolů. Jeho stisknutím rozbalíme seznam „historií“, tedy hodnot, které zde byly zadány předtím. Za některými z polí je tlačítko se třemi tečkami. Jeho stisknutím vyvoláme další dialogové okno, ve kterém můžeme měnit pořadí položek nebo přidávat nové. Totéž ovšem můžeme dělat i „ručně“, přímo v jednotlivých vstupních polích.

Version Info

Na této kartě určujeme, zda se mají do výsledného souboru uložit informace o číslu verze a které to mají být. Hovořili jsme o ní podrobně v kap. 6.3, kde také najdete její obrázek (obr. 6.8).

Packages

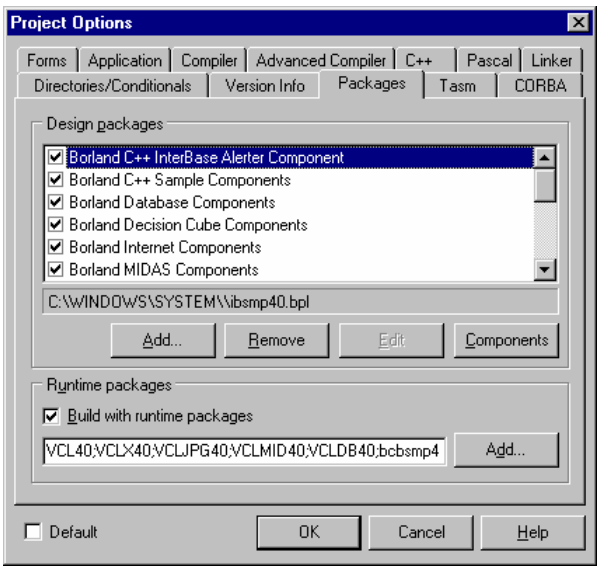
Na této kartě předepisujeme způsob zacházení s balíčky (package), což jsou vlastně dynamické knihovny obsahující skupiny komponent.

Ve skupině Design packages určujeme, které balíčky mají být k disposici v době návrhu programu. V seznamu se zaškrtávacími poli jsou jejich názvy, ve stavové řádce pod ním pak jména odpovídajících souborů. Balíčky můžeme přidávat nebo odebírat zaškrtnutím nebo pomocí tlačítek Add... a Remove. Tlačítko Edit otevře vybraný balíček v editoru balíčků, pokud je k disposici zdrojový kód. Tlačítko Components otevře okno se seznamem komponent, které jsou v něm k disposici, a jejich ikon.

Ve skupině Runtime packages určujeme, které balíčky se mají použít při vytváření výsledného programu. Zaškrtnutím pole Build with runtime packages předepisujeme, že se mají použít balíčky

(připomeňme si, že to jsou dynamické knihovny, a proto je musíme dodávat s programem). Necháme-li toto pole volné, připojí se komponenty jako statická knihovna.

Nové balíčky přidáme tlačítkem Add... nebo tím, že je zapíšeme do seznamu v pole, které je vedle tohoto tlačítka; jednotlivé položky jsou jako obvykle odděleny středníky.



Obr. 12.9 Práce s balíčky

Tasm, CORBA

Karta Tasm obsahuje volby pro Turbo Assembler, karta CORBA obsahuje volby pro distribuované aplikace vytvářené pomocí tohoto průmyslového standardu. Obě tato témata přesahují rámec naší knihy, a proto se jimi zde nebudeme zabývat. Jejich význam najdete v případě potřeby v nápovědě.

12.2 Vizuální návrh programu

Okno s vizuálním návrhem programu je k disposici pouze při návrhu aplikací založených na knihovně VCL. O práci s ním jsme podrobněji hovořili v kapitole 1, zde si uvedeme pouze přehled některých běžných operací.

Vložení komponenty z palety: Kliknutím na záložku palety vybereme odpovídající paletu, kliknutím na ikonu komponenty vybereme požadovanou komponentu. Kliknutím do okna vizuálního návrhu komponentu umístíme. Vizuální komponenty lze dále myší upravovat (měnit velikost, polohu).

Vložení několika instancí téže komponenty: Při výběru komponenty stiskneme zároveň klávesu SHIFT. Ikona komponenty se zvýrazní (dostane modrý rámeček). Každé následující kliknutí do okna vizuálního návrhu sem vloží jednu instanci zvolené komponenty.

Zrušení výběru komponenty na paletě: Klikneme na ikonu se šipkou na levé straně palety.

Odstranění komponenty z vizuálního návrhu: Kliknutím ji vybereme a stiskneme klávesu DEL. Pak je ovšem potřeba ručně odstranit ze zdrojového textu všechny odkazy na ni.

Přidání nového okna k projektu: Příkazem File | New... vyvoláme zásobník objektů a v něm zvolíme Form. Můžeme také použít tlačítko New Form na nástrojovém panelu. Tím se zdrojový a hlavičkový soubor s definicí třídy okna a soubor .dfm s popisem vlastností okna.

Odstranění okna z projektu: Musíme odstranit celý modul, tj. soubor .cpp, .h a .dfm. Příkazem View | Project manager vyvoláme správce projektů, vybereme příslušný modul a stiskneme DEL nebo tlačítko Remove. Můžeme také použít příkaz Project | Remove from Project....

Podobně odstraníme z projektu i modul, který nemá své okno.

Textový popis okna a komponent v něm: Kliknutím pravým tlačítkem myši na vizuálním návrhu okna vyvoláme příruční nabídku a v ní vybereme příkaz View as text. Zpět se přepneme podobně, pouze zvolíme příkaz View as Form. K přepínání tam i zpět lze použít i klávesovou zkratku ALT+F12.

Zobrazení skrytého okna: Příkazem View | Forms..., tlačítkem View Forms nebo klávesovou zkratkou CTRL+F12 vyvoláme dialogové okno View Form, ve kterém zvolíme potřebné okno.

Položení vizuálního návrhu okna před editor zdrojového kódu nebo naopak: Klávesová zkratka F12.

Úprava vlastností komponent v době návrhu: Vybereme komponentu (myší ve vizuálním návrhu nebo v rozbalovacím seznamu v záhlaví okna inspektora objektů). V okně inspektora objektů vyhledáme v levé polovině okna odpovídající vlastnost a v pravé polovině okna zapíšeme její hodnotu.

V některých případech se po vybrání vlastnosti kliknutím myší na její jméno objeví vedle pole s hodnotou tlačítko. Obsahuje-li šipku dolů, objeví se po kliknutí na ně seznam možných hodnot, ze kterých si vybereme. (V některých případech nemáme jinou možnost, v některých případech můžeme zapsat i jinou hodnotu.)

Obsahuje-li toto tlačítko tři tečky, vyvoláme jeho stisknutím editor této vlastnosti (dialogové okno pro výběr fontu, dialogové okno pro zadávání seznamu řetězců apod.).

Vlastnosti komponent můžeme upravovat i v textovém zobrazení návrhu okna.

12.3 Editor zdrojového textu

Po spuštění je okno editoru zdrojového textu skryto pod vizuálním návrhem okna. Pořadí okna s vizuálním návrhem a okna editoru lze zaměnit stisknutím klávesy F12. Editor C++ Builderu umí i české znaky, je-li na počítači nastaveno české národní prostředí. Ve starších verzích je třeba nastavit české písmo (Okno Tools | Environment Options..., karta Display, pole Editor font).

Editor C++ Builderu umí zvýrazňovat syntaktické kategorie položek zdrojového textu v jazycích C, C++, Pascal a SQL. Použitý jazyk rozlišuje podle přípony.

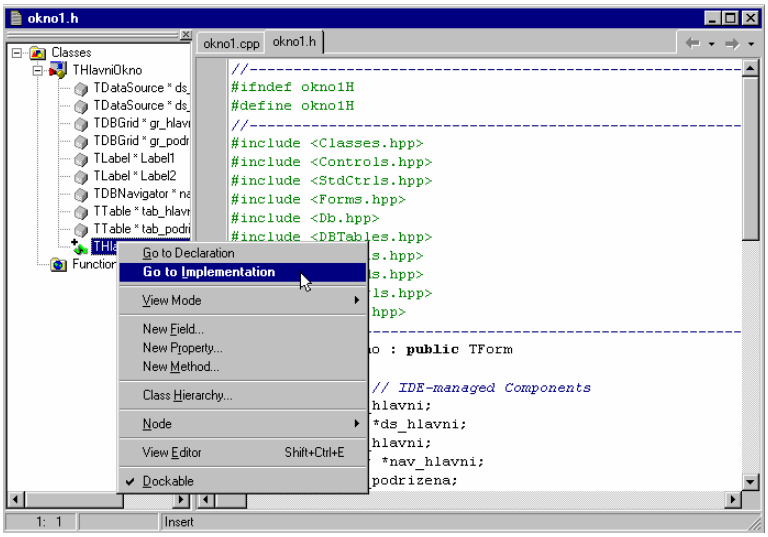
V tomto editoru lze používat mnohé klávesové zkratky známé z předchozích verzí borlandských vývojových prostředí. Můžeme v něm také používat schránku Windows obvyklým způsobem.

Běžné editovací operace

Orientace ve zdrojovém textu

Průzkumník tříd (Class Explorer): Je nejmocnějším nástrojem pro orientaci ve zdrojovém textu. Jeho okno je připojeno k levé straně okna se zdrojovým textem programu, lze je ale v případě potřeby oddělit a přemístit nebo uzavřít. Tento průzkumník ukazuje strom zobrazující aktuální strukturu projektu z hlediska zdrojového kódu. Jeho základními uzly jsou jednotlivé třídy a globální funkce. Rozvinutím uzlů zobrazujících třídy pak uvidíme jednotlivé složky (metody, vlastnosti i datové složky).

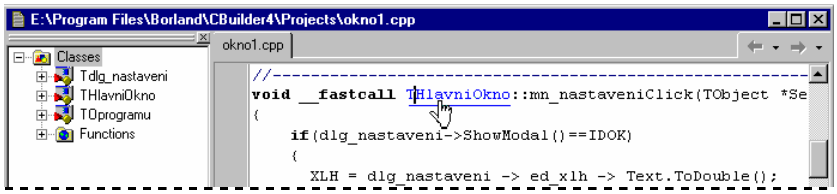
Průzkumník tříd umožňuje snadno vyhledat ve zdrojovém textu deklaraci či implementaci třídy, funkce, metody, vlastnosti atd. Stačí vybrat myší hledanou funkci, třídu nebo její složku, pravým tlačítkem myši vyvolat příruční nabídku a v ní použít příkaz Go to Declaration, který nás přenese na odpovídající místo hlavičkového souboru, nebo Go to Implementation, který nás přenese na odpovídající místo zdrojového souboru, a to i do zdrojových souborů standardních komponent (napsaných v Pascalu).



Obr. 12.10 Vyhledání implementace metody pomocí průzkumníka tříd

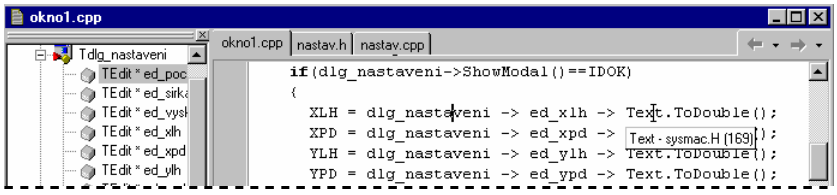
Prohlížeč kódu (code browser): Hodí se, např. když chceme získat podrobnější informace o komponentách. Stačí kliknout pravým tlačítkem myši na identifikátor a v příruční nabídce zvolit Find Declaration a prostředí nás za příznivých okolností přenese do odpovídajícího hlavičkového souboru.

Můžeme také využít skutečnosti, že identifikátory ve zdrojovém textu mohou fungovat jako hyperlinky. Stačí stisknout klávesu CTRL a ukázat kurzorem myši na identifikátor. Vybraný identifikátor se změní v hyperlink a stisknutím levého tlačítka se přeneseme do souboru s deklarací. (Poslední dvě možnosti fungují zejména pokud jde o vizuální komponenty. Prostředí zobrazí soubor .HPP, vytvořený překladačem Pascalu pro použití v C++.)



Obr. 12.11 Identifikátor jako hyperlink: Stačí stisknout Ctrl a ukázat

Kde je identifikátor definován: Stačí-li nám tato informace, ukážeme prostě kurzorem myši na identifikátor. Po chvilce se v bublině objeví jméno odpovídajícího hlavičkového souboru a číslo řádky.



Obr. 12.12 Bublinová nápověda o místě definice identifikátoru

Blokové operace

Označení bloku: Tažením myší. Můžeme také umístit kurzor na začátek, stisknout klávesu SHIFT a pomocí kláves s kurzorovými šipkami přemístit kurzor na konec bloku. Blok se chová jako v běžných editorech pod Windows.

Zrušení bloku: Kliknutím myší mimo blok nebo stisknutím některé z kláves, které přemísťují kurzor. Pokud je na kartě Tools | Environment Options nastavena volba Persistent blocks (trvalé bloky), použijeme klávesovou zkratku CTRL+K, H. [63]

[63] Zápis CTRL+K, H znamená, že nejprve stiskneme současně CTRL+K a pak stiskneme H. Podobně zapisujeme i jiné složitější klávesové zkratky.

Přemístění označeného bloku: Přetažením myší.

Vymazání označeného bloku: Stisknutím klávesy DEL.

Změna odsazení označeného bloku: Posun vpravo CTRL+K, I posun vpravo CTRL+K, U.

Uložení označeného bloku do souboru, resp. přečtení bloku ze souboru: CTRL+K, W, resp. CTRL+K, R.

Editování zdrojového textu

Odstranění řádky: Použijeme klávesovou zkratku CTRL+Y.

Odstranění textu od kurzoru do konce slova: CRTL+T**.**

Odstranění textu od kurzoru do konce řádky: CTRL+Q, Y.

Vrácení chybné operace, zopakování předchozí operace: Použijeme příkazy Edit | Undo, resp. Edit | Redo, příp. klávesové zkratky CTRL+Z, resp. CTRL+SHIFT+Z. Poznamenejme, že lze vrátit až 32767 editovacích operací ve zdrojovém textu. Operace Zpět se však nedá použít pro operace s komponentami ve vizuálním návrhu.

Nástroje, které usnadňují práci se zdrojovým kódem

Vytvoření handleru: V inspektoru objektů vybereme komponentu, o jejíž událost nám jde, přejdeme na stránku Events (události) a vybereme událost, kterou chceme ošetřit. Po dvojkliku na toto pole nás prostředí přenese do zdrojového souboru, ve kterém zároveň vytvoří kostru handleru (hlavičku se správnými parametry a závorky { a } uzavírající tělo). Zároveň vloží deklaraci handleru i do sekce __published deklarace třídy okna.

Nic nám samozřejmě nebrání napsat si handler úplně sami, musíme jej ale vložit do sekce public.

Připojení existujícího handleru k události: V inspektoru objektů na kartě Events vybereme událost a kliknutím rozvineme seznam dostupných handlerů. Chceme-li určit handler za běhu programu, prostě ho události přiřadíme, např. příkazem

```
OnClick = tlacitkoClick;
```

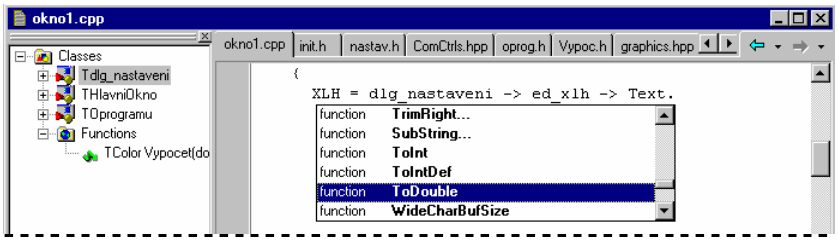
Identifikátor události totiž představuje ukazatel na handler (ukazatel typu __closure).

Odpojení handleru od události (zrušení ošetřování události): V inspektoru objektů na kartě Events vybereme událost a v odpovídajícím poli smažeme jeho jméno. Za běhu programu můžeme zrušit ošetřování události tím, že události přiřadíme hodnotu 0, např.

```
OnClick = 0;
```

Zrušení handleru: Chceme-li odstranit handler ze zdrojového textu, smažeme jeho tělo a uložíme soubor. Prostředí se postará o zbytek, tj. odstraní kostru a deklaraci z těla třídy.

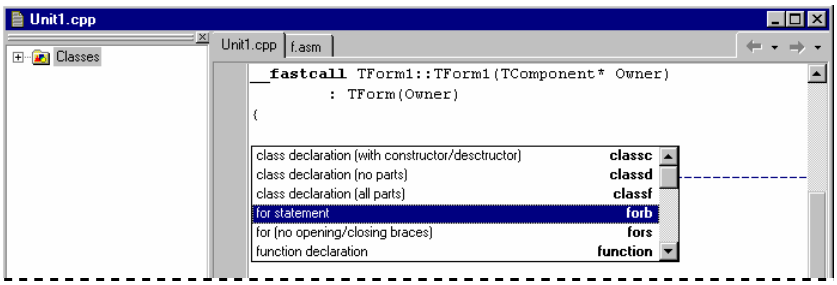
Dokončování kódu: Napíšeme-li jméno funkce a odpovídající závorku, nabídne nám prostředí v bublinové nápovědě seznam typů parametrů. Napíšeme-li jméno instance objektového typu a ze ním operátor tečka nebo operátor ->, nabídne nám prostředí seznam složek. V něm si můžeme vybrat složku, kterou chceme, a stisknutím mezery ji vložit do zdrojového textu.



Obr. 12.13 Dokončování kódu

Code Insight: Tento nástroj usnadňuje psaní běžných konstrukcí jazyka C++. Po stisknutí klávesové zkratky CTRL+J se v místě kurzoru objeví okénko, ve kterém si můžeme vybrat jednu z konstrukcí,

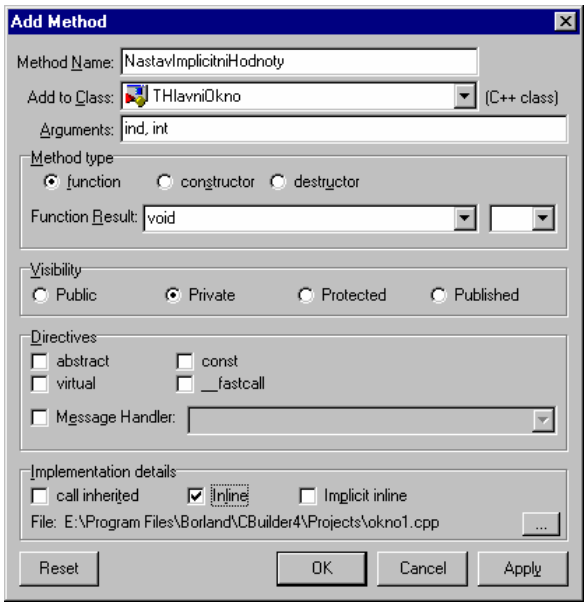
které chceme do zdrojového textu vložit. Např. na obr. 12.10 se po stisknutí klávesy ENTER vloží do zdrojového textu kostra příkazu for.



Obr.12.14 Code Insight

Přidávání složek do tříd pomocí průzkumníka tříd: Klikneme pravým tlačítkem myši v okně průzkumníka tříd a v příruční nabídce zvolíme jeden z příkazů New Field... (nová datová složka), New Property... (nová vlastnost) nebo New Method... (nová metoda). Objeví se dialogové okno, ve kterém zadáme potřebné informace.

Dialogové okno pro přidání metody ukazuje obr. 12.15.



Obr. 12.15 Přidáváme novou metodu do třídy

V poli Method Name zadáme identifikátor nové metody. V poli Add to Class nám průzkumník tříd nabídne právě vybranou třídu. Chceme-li metodu přidat do jiné, můžeme si vybrat z rozbalovacího seznamu našich tříd v projektu. V poli Arguments zadáme typy formálních parametrů oddělené čárkami.

Ve skupině Method type zvolíme, zda jde o konstruktor, destruktor nebo funkci, a v poli Function Result uvedeme typ výsledku. Můžeme ho zapsat „ručně“ nebo můžeme použít připojený seznam standardních typů. V následujícím nepojmenovaném poli můžeme zadat jeden ze znaků * nebo &, pokud má funkce vracet ukazatel nebo referenci.

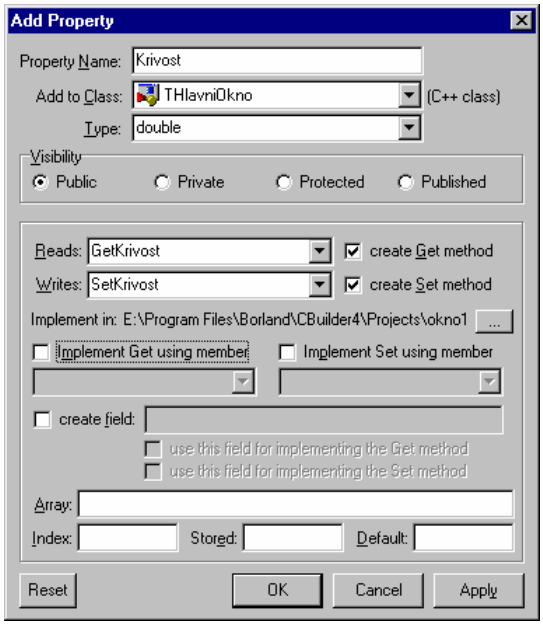
Ve skupině Visibility zvolíme přístupová práva pro novou funkci.

Zaškrtávací políčka ve skupině Directives umožňují předepsat, zda má tato metoda být čistě virtuální (abstract), virtuální (virtual), zda má jít o metodu pro konstantní objekty (const) a zda má používat registrovou volací konvenci (__fastcall). Poslední políčko, Message handler, umožňuje určit, že jde o funkci, která bude určena ke zpracování zpráv od Windows. Pokud je zaškrtneme, budeme moci v následujícím seznamu určit, na kterou zprávu má reagovat.

Ve skupině Implementation details určujeme, zda má jít o vloženou funkci (inline), zda má jít o metodu deklarovanou v těle třídy v hlavičkovém souboru (Implicit inline) a zda má volat stejnojmennou metodu předka (Call inherited). Pod všemi poli je vypsána cesta a jméno souboru, ve kterém budou změny provedeny.

Po stisknutí OK vloží prostředí deklaraci nové metody do těla třídy a vytvoří kostru implementace. Pokud jsme si poručili funkci pro zpracování zpráv od Windows, vloží do definice třídy také makra, která tuto metodu „napojí“ na odpovídající zprávu.

Vkládání datové složky je podstatně jednodušší, neboť zde zadáváme pouze jméno, typ a přístupová práva (a samozřejmě třídu, do které novou složku přidáváme).



Obr. 12.16 Vkládání nové vlastnosti

Při vkládání nové vlastnosti dostaneme okno, ve kterém kromě jména a typu, jména, třídy a přístupových práv zadáváme také způsob ukládání a čtení. V polích Reads a Writes můžeme uvést jména existujících metod, které mají sloužit ke čtení, resp. k uložení hodnoty vlastnosti. Pokud místo toho zaškrtneme pole create Set method, resp. create Get method, vytvoří prostředí zároveň i kostry metod pro nastavení, resp. čtení hodnoty vlastnosti. Políčka Implement ... using member umožňují specifikovat datovou složku, která se má k implementaci těchto metod využít. Zaškrtneme-li pole Create field, vytvoří prostředí novou datovou složku, do níž se bude hodnota vlastnosti ukládat. Pole Array, Index, Stored a Default slouží ke specifikaci indexovaných vlastností, jejich indexů, specifikace způsobu ukládání a implicitní hodnoty.

Práce se soubory

Otevření souboru: Příkaz File | Open nebo tlačítko Open na nástrojovém panelu. Lze také vyhledat soubor ve správci projektů a dvojklikem ho otevřít. Nový soubor se otevře na samostatné kartě.

Otevření hlavičkového souboru, je-li otevřen zdrojový soubor (nebo naopak): Je-li zdrojový soubor aktivní, klikneme v jeho okně pravým tlačítkem myši a z příruční nabídky použijeme příkaz Open Source/Header File nebo klávesovou zkratku CTRL+F6. Je-li požadovaný soubor již otevřen, prostředí přejde na jeho kartu, není-li otevřen, otevře ho a přejde na jeho kartu.

Otevření souboru, se kterým jsme nedávno pracovali: Příkaz File | Reopen. Ve spodní části podřízené nabídky, kterou tímto příkazem vyvoláme, je seznam nedávno použitých souborů, ve kterém si můžeme vybrat.

Vytvoření nového zdrojového souboru: S novým oknem, vláknem nebo datovým modulem se nový zdrojový a hlavičkový soubor vytvoří automaticky. Jinak použijeme File | New... a zvolíme Unit. Vytvořený modul bude součástí projektu.

Pokud chceme jen samostatný hlavičkový soubor, nebo pokud chceme např. vytvořit soubor .RC, použijeme File | New... a zvolíme Text. Při uložení soubor pojmenujeme podle potřeby a pak ho připojíme k projektu (viz výše).

Uzavření souboru: Příkaz Close Page z příruční nabídky nebo klávesová zkratka CTRL+F4. Pozor: Příkaz File | Close uzavře aktivní zdrojový soubor spolu s hlavičkovým souborem.

Uzavření všech souborů a celého projektu: Příkaz File | Close All....

Vyhledávání a nahrazování

Vyhledávání textu: Příkazem Search | Find nebo klávesovou zkratkou CTRL+F vyvoláme dialogové okno Find Text s aktivní kartou Find (obr. 12.17). Ve vstupním poli Text to find zadáme hledaný text. Další přepínače a volby určují způsob vyhledávání.

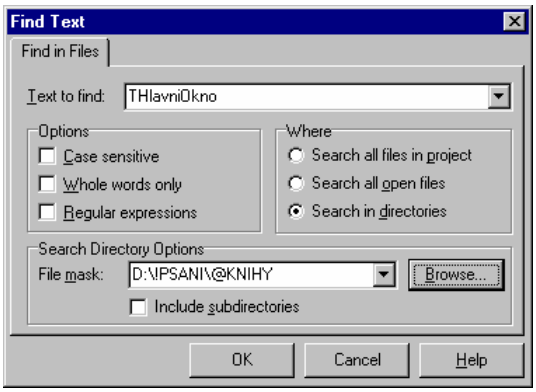
Zaškrtneme-li pole Case sensitive, budou se při vyhledávání rozlišovat malá a velká písmena. Možnost Whole words only předepisuje brát v úvahu pouze celá slova a zaškrtnutím Regular expressions dovolíme používat ve specifikaci vyhledávaného textu např. žolík * nahrazující libovolný úsek textu, speciální znaky určující začátek nebo konec řádku atd. (Podrobnosti najdete v nápovědě, kterou vyvoláte stisknutím tlačítka Help v tomto okně.)

Obr. 12.17 Vyhledávání textu

Přepínače v poli Scope určují, zda se bude zadaný text hledat v celém souboru (Global) nebo jen v aktuálním výběru (Selected text). Přepínače v poli Direction určují směr prohledávání — směrem ke konci souboru (Forward) nebo obráceně (Backward) — a přepínače v poli Origin určují, kde prohledávání začne zda od kurzoru (From cursor) nebo od počátku, resp. konce souboru (Entire scope).

Opakované vyhledávání: Chceme-li zopakovat předchozí vyhledávání, použijeme příkaz Search | Search Again nebo klávesovou zkratku F3.

Prohledávání souborů: Příkazem Search | Find in Files vyvoláme dialogové okno Find text, které bude obsahovat pouze kartu Find in Files (obr. 12.18). Vstupní pole Text to find a volby ve skupině Options mají stejný význam jako tytéž položky na kartě Find.



Obr. 12.18 Prohledávání souborů

Přepínače ve skupině Where určují, zda se mají prohledávat soubory aktuálního projektu (Search all files in project), otevřené soubory (Search all open files) nebo soubory v daném adresáři (Search in directories). Zvolíme-li poslední možnost, můžeme v poli File mask zadat adresář, ve kterém chceme hledat;

zaškrtnutím pole Include subdirecories přikážeme prohledat i podadresáře zvoleného adresáře. (Jde o variantu známého programu grep s upraveným uživatelským rozhraním.)

Nahrazování: Příkazem Search | Replace nebo klávesovou zkratkou CTRL+R vyvoláme dialogové okno Replace text, které se od okna Find Text liší ve třech ohledech: Obsahuje navíc vstupní pole Replace with, ve kterém uvedeme řetězec, kterým se má nalezený text nahradit, zaškrtávací pole Prompt on replace, které způsobí, že se nás IDE před každou náhradou zeptá na souhlas, a tlačítko Replace All, které způsobí, že IDE nahradí bez ptaní všechny výskyty zadaného řetězce.

12.4 Ladění

Vzhledem k rozsahu a zaměření této knihy zde nebudeme hovořit vůbec o ladění distribuovaných aplikací, o ladění více procesů a o ladění vzdálených procesů.

Základní možnosti jsou podobné jako v předchozích verzích borlandských překladačů; kombinují v sobě schopnosti integrovaného ladicího programu a Turbo Debuggeru, který byl součástí některých předchozích borlandských produktů.

Krokování, zarážky

Základním nástrojem pro ladění programu je krokování, při kterém provádíme program příkaz po příkazu. Prostředí při krokování označí příkaz, který se provede jako příští, barevným pruhem, a řádky zdrojového textu, které je možné krokovat, barevnými body.[64]

Do zdrojového programu můžeme vkládat zarážky (breakpointy), místa, na kterých se běžící program za jistých okolností zastaví.

Krokování se vstupem do funkce: Příkaz Run | Trace into nebo klávesová zkratka F7 způsobí, že program přejde na následující příkaz a bude čekat. Je-li to volání funkce, vstoupí do ní, tj. přejde na její první příkaz.

Jestliže tímto příkazem ladění začneme, bude prostředí krokovat úvodní operace (startovací kód programu, vytváření oken atd.). Protože pro tyto části však není k dispozici zdrojový kód, bude krokovat disasemblovaný strojní kód.

Krokování bez vstupu do funkce: Příkaz Run | Step over nebo klávesová zkratka F8 způsobí, že program přejde na následující příkaz a bude čekat. Je-li to volání funkce, provede její tělo jako jediný příkaz, tj. nebude do ní vstupovat.

Doběhnutí ke kurzoru: Nastavíme kurzor na řádek, na kterém se má program zastavit, a použijeme příkaz Run | Run to Cursor nebo klávesovou zkratku F7. Program se rozeběhne od počátku nebo z aktuální pozice a doběhne k řádce s kurzorem; příkaz na této řádce již neprovede.

Vyběhnutí z funkce: Příkazem Run | Run Until Return nebo klávesovou zkratkou SHIFT+F8 způsobíme, že program doběhne do konce aktuální funkce, vrátí se z ní a zastaví se na prvním příkazu za voláním aktuální funkce.

Návrat k aktuálnímu příkazu: Při krokování se může stát, že si potřebujeme prohlédnout i jiné části programu než tu, která se právě provádí. Chceme-li pak rychle najít aktuální příkaz, použijeme příkaz Run | Show Execution Point.

Konec krokování: použijeme příkaz Run | Program Reset nebo klávesovou zkratku CTRL+F2.

Přerušení programu: Příkaz Run | Program Pause.

Spuštění programu: Příkaz Run | Run nebo klávesová zkratka F9.

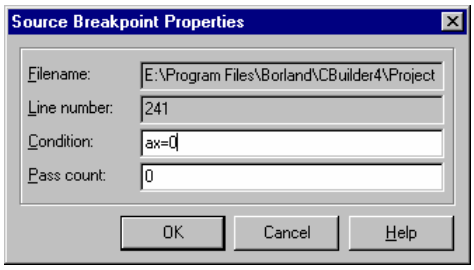
Parametry programu (příkazová řádka): příkazem Run | Parameters... vyvoláme dialogové okno Run Parameters, které má dvě karty — Local a Remote. Karta Remote se týká ladění vzdálených aplikací, proto ji pomineme. Na kartě Local zadáme ve vstupním poli Parameters parametry příkazové řádky programu.

[64] Při krokování se jako příkaz chová také hlavička funkce a závorka } uzavírající tělo funkce. Hlavička představuje standardní ošetření zásobníku při vstupu do funkce, uzavírací složená závorka představuje standardní výstupní ošetření zásobníku.

Vstupní pole Host Application umožňuje zadat „hostitelskou“ aplikaci pro ladění dynamických knihoven.

Vložení zarážky (breakpointu) do zdrojového programu: Dvojklikem myší na svislý šedý pruh na levé straně okna se zdrojovým programem vložíme do programu „prostou“ zarážku, na které se program zastaví vždy, když na ni narazí. Řádek se zarážkou prostředí označí červeným kroužkem a červeným pruhem (obr. 12.20).

Chceme-li změnit vlastnosti zarážky, vyvoláme stisknutím pravého tlačítka myši na červeném kruhu označujícím zarážku příruční nabídku a z ní příkazem Breakpoint Properties... stejnojmenné dialogové okno (obr. 12.19).



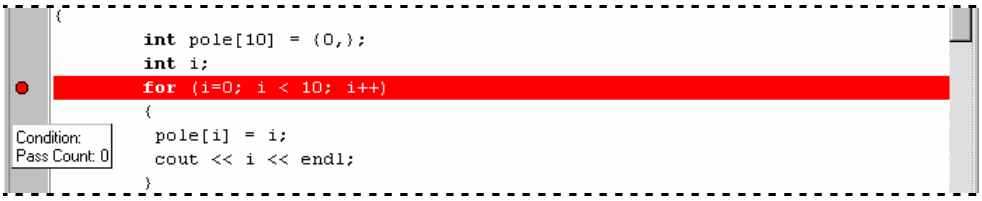
Obr. 12.19 Vlastnosti zarážky

V tomto okně ukazují první dva řádky jméno souboru, do kterého jsme zarážku vložili, a číslo řádky. Na dalších dvou můžeme zadat podmínku přerušení (Contition) počet průchodů před přerušením (Pass count). K přerušení dojde, nastane-li alespoň jedna z uvedených okolností; přitom počet průchodů je třeba zadat vždy.

Pro vložení zarážky tohoto druhu můžeme také použít příkaz Run | Add Breakpoint | Source Breakpoint. Objevím se dialogové okno prakticky shodné s oknem na obr. 12.19, ve kterém zadáme číslo řádky, podmínky a počet průchodů.

Zjištění vlastností zarážky: Stačí umístit kurzor myši na červený kroužek označující zarážku. Po chvilce se v bublinové nápovědě objeví podmínky, na které je zarážka vázána.

Můžeme také vyvolat okno Breakpoint Properties (obr. 12.19), viz předchozí odstavec.



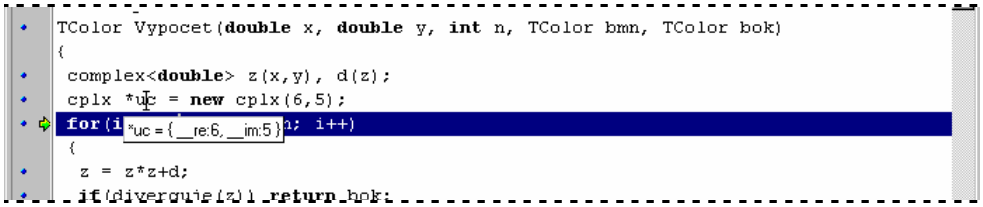
Obr. 12.20 Vlastnosti zarážky v bublinové nápovědě

Zarážka vázaná na adresu ve zdrojovém kódu: Lze ji zadat jen za běhu programu. Příkazem Run | Add Breakpoint | Address Breakpoint... vyvoláme dialogové okno Address Breakpoint, ve kterém zadáme adresu zjištěnou v okně CPU. I zde můžeme zadat podmínku a počet průchodů.

Zarážka vázaná na změnu dat: Lze ji zadat jen za běhu programu. Příkazem Run | Add Breakpoint | Data Breakpoint... vyvoláme dialogové okno Data Breakpoint, ve kterém zadáme adresu proměnné (lze ji zadat i výrazem tvaru &i). I zde můžeme zadat podmínku a počet průchodů.

Vyřazení zarážky: Opotřebujeme-li zarážku na čas vyřadit a nechceme-li ji odstranit, vyhledáme ji, nastavíme na ni kurzor, vyvoláme příruční nabídku a v ní zrušíme zaškrtnutí položky Enable. Podobným způsobem zarážku zase zprovozníme.

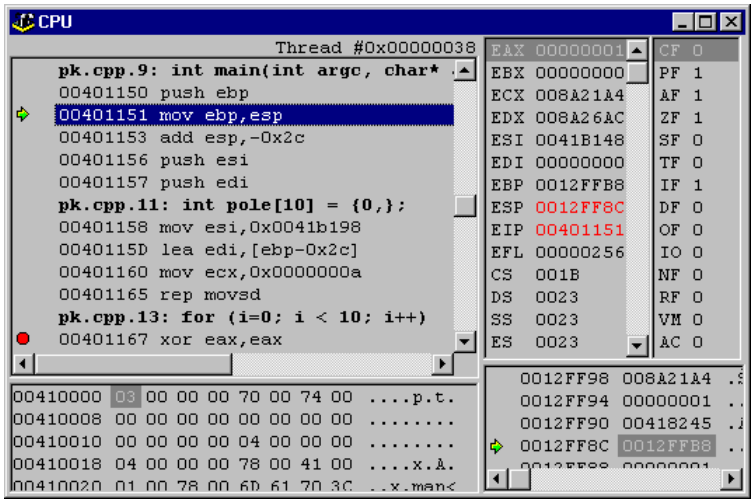
Rychlé zjištění hodnoty proměnné: Ukážeme-li při krokování kurzorem myši na identifikátor proměnné, objeví se v bublinové nápovědě její hodnota (obr. 12.21).



Obr. 12.21 Hodnota proměnné v bublinové nápovědě

Krokování ve strojovém kódu (okno CPU): Za běhu programu vyvoláme okno CPU příkazem View | Debug Windows | CPU (obr. 12.22) nebo klávesovou zkratkou CTRL+ALT+C. Objeví se také, jestliže zahájíme ladění příkazem Run | Trace into nebo klávesovou zkratkou F7.

Toto okno obsahuje 5 dílčích oken. První (vlevo nahoře) obsahuje právě prováděné instrukce ve strojním kódu a vedle nich jejich zpětný překlad do asembleru. Tučně jsou zde odpovídající řádky zdrojového kódu. Aktuální instrukce je vyznačena modrým pruhem a zelenou šipkou, zarážky červeným kolečkem před kódem. Při krokování v tomto okně postupuje prostředí po jednotlivých instrukcích strojního kódu. I zde lze použít stejné příkazy jako při krokování ve zdrojovém programu.



Obr. 12.22 Krokování ve strojním kódu

Druhé pole (uprostřed nahoře) obsahuje aktuální hodnotu uložené v registrech procesoru. Hodnoty, které se při poslední instrukci změnily, jsou barevně odlišeny.

Třetí pole (vpravo nahoře) obsahuje hodnotu jednotlivých příznaků, tedy bitů z registru EFLAGS.

Čtvrté pole (vlevo dole) obsahuje výpis části datového segmentu; po levé straně je vždy adresa, pak hexadecimální výpis skupiny bajtů a nakonec výpis těchže bajtů ve znakové podobě.

Páté pole (vpravo dole) obsahuje výpis dat uložených na vrcholu zásobníku. (v levé části je vždy adresa, pak následuje hodnota. Zelená šipka označuje vrchol zásobníku.

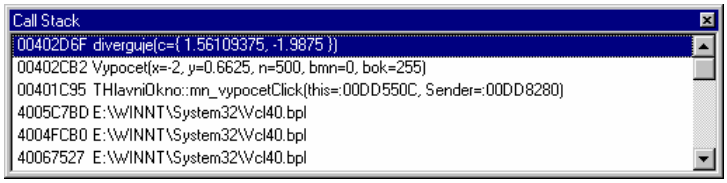
Příruční nabídka v tomto okně umožňuje vyhledat zadanou adresu, měnit data v paměti, vkládat a rušit zarážky, měnit příznaky, přejít do jiného vlákna atd.

Pomocná okna

Tato okna usnadňují ladění a umožňují orientovat se ve zdrojovém textu, v zarážkách atd. Většinu z nich lze uzavřít stisknutím klávesu ESC.

Seznam zarážek: Příkazem View | Debug Windows | Breakpoints... nebo klávesovou zkratkou CTRL+ALT+B vyvoláme okno obsahující seznam zarážek a jejich charakteristiky (podmínky ap.).

Posloupnost volání funkcí: Příkazem View | Debug Windows | Call stack... nebo klávesovou zkratkou CTRL +ALT+S vyvoláme okno obsahující posloupnost volání funkcí od startovacího kódu až po právě aktivní funkci a jejich parametry včetně jmen (pokud jsou pro dotyčnou funkci k dispozici ladicí informace). V první řádce je právě aktivní funkce. Příkazy z příruční nabídky můžeme snadno přejít do zdrojového kódu vybrané funkce.



Obr. 12.23 Posloupnost volání funkcí

Sledované výrazy: Chceme-li při krokování sledovat hodnoty některých proměnných nebo výrazů, použijeme okno Watch list. Vyvoláme ho příkazem View | Debug Windows | Watches... nebo klávesovou zkratkou CTRL +ALT+W. Je-li toto okno aktivní, objeví se po stisknutí klávesy INS dialogové okno Watch properties, ve kterém v poli Expression zadáme požadovaný výraz.

Chceme-li změnit sledovaný výraz, vybereme jej a stiskneme klávesu ENTER. Objeví se okno Watch properties, ve kterém výraz upravíme.

Chceme-li sledovaný výraz z tohoto okna odstranit, vybereme jej a stiskneme klávesu DEL.

Lokální proměnné: Ke sledování hodnot lokálních proměnných v aktuálním kontextu nejlépe poslouží okno Local Variables, které vyvoláme příkazem View | Debug Windows | Local Variables... nebo klávesovou zkratkou CTRL +ALT+L.

Stav vláken: Aktuální informace o stavu vláken (threadů) zjistíme z okna Thread Status, které vyvoláme příkazem View | Debug Windows | Threads... nebo klávesovou zkratkou CTRL +ALT+T.

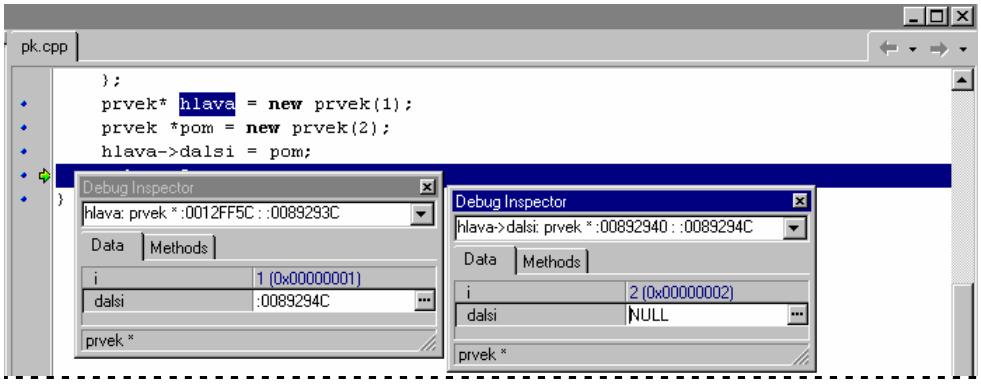
Přehled modulů v paměti: Příkazem View | Debug Windows | Modules... nebo klávesovou zkratkou CTRL +ALT+M vyvoláme okno obsahující přehled modulů (dynamických knihoven, balíčků ap.) aktuálně zavedených do paměti.

Přehled událostí: Příkazem View | Debug Windows | Event log... nebo klávesovou zkratkou CTRL +ALT+E vyvoláme okno Event log (deník událostí) obsahující přehled událostí od spuštění programu. Pod událostmi zde rozumíme zavedení modulu do paměti, přechod zarážky atd.

Inspekční okno (ladicí inspektor): Toto okno oceníme zejména při ladění programů obsahujících dynamické datové struktury. Vyvoláme ho za běhu programu příkazem Run | Inspect; v dialogovém okně, které se objeví, zadáme jméno proměnné, kterou chceme podrobněji prozkoumat. Okno ladicího inspektora (obr. 12.24) ukazuje typ, adresu a aktuální hodnotu proměnné. V případě instancí objektových typů ukazuje navíc seznam metod a v případě komponent též hodnoty vlastností.

V případě pole ukáže jednotlivé prvky, v případě ukazatele také hodnotu, na kterou ukazuje. Je-li mezi složkami zobrazené proměnné ukazatel, můžeme ho vybrat a stisknout klávesu ENTER; prostředí zobrazí další inspekční okno, ve kterém uvidíme obsah proměnné, na kterou tento ukazatel ukazoval. Tímto způsobem můžeme postupně projít i několik členů seznamu nebo jiné složitější datové struktury.

Stisknutím tlačítka se třemi tečkami vedle hodnoty vyvoláme dialogové okno, které umožňuje změnit hodnotu zkoumané proměnné.



Obr. 12.24 procházíme seznam pomocí ladicího inspektora

Kalkulačka, změna hodnoty proměnné za běhu: Příkazem Run | Evaluate/Modify nebo klávesovou zkratkou CTRL+F7 vyvoláme příruční kalkulačku, která umožňuje používat i proměnné z právě běžícího programu. Jestliže v poli Expression zadáme výraz a stiskneme ENTER, objeví se v poli Result

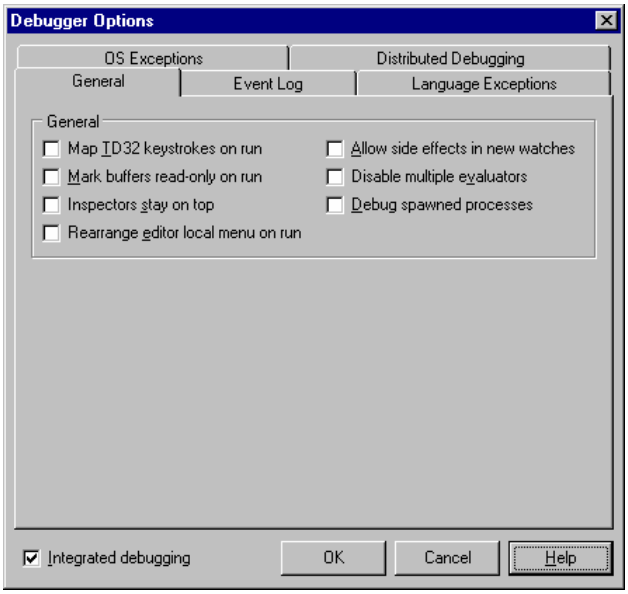
výsledek. Jestliže v poli Expression zadáme výraz, který představuje l-hodnotu, můžeme v poli New value zadat novou hodnotu.

Volby pro ladění

Příkazem Tools | Debugger Options... vyvoláme dialogové okno Debugger Options (obr. 12.25). V něm na kartě General najdeme volby, které se týkají obecného chování debuggeru. První dvě zaškrtávací políčka Map TD32 keystrokes on run a Mark buffers read-only on run, nastavují chování známé z 32bitové verze Turbo Debuggeru (stejné klávesové zkratky, zakazují editování). Další volba, Inspectors stay on top, určuje, zda budou pomocná ladicí okna stále na vrchu, nad ostatními okny. Volba Allow side effects in new watches, určuje, zda se vyhodnotí sledovaný výraz, i když jeho výpočet způsobí vedlejší efekt. Možnost Disable multiple evaluators zakazuje používání pascalských vyhodnocovacích nástrojů v C++ Builderu. Zaškrtneme-li pole Debug spawned processes, pak pokud náš program spustí nějaký další proces, automaticky ho budeme také ladit.

Na kartě Event Log najdeme volby, které určují chování deníku. Volba Clear log on run způsobí, že se obsah deníku před začátkem každého pokusu smaže. Volba Unlimited Length určuje, zda bude délka deníku omezena. Volby ve skupině Messages určují, jaké zprávy se budou do deníku zaznamenávat. Mohou to být zprávy o zarážkách (Breakpoint messages), o chodu procesu (Process messages), zprávy způsobené voláním funkce OutputDebugString (Output messages) a zprávy od Windows (Windows messages).

Na kartě Language Exceptions určujeme, které typy výjimek se mají při ladění ignorovat (pole Exception Types to Ignore) a zda se má ladění přerušit při výjimce z Delphi a z jazyka C++ (pole Stop on Delphi Exceptions a Stop on C++ Exceptions).



Obr. 12.25 Nastavené ladicího programu

Na kartě OS Exceptions určujeme, jak má ladicí program reagovat na výjimky operačního systému (porušení ochrany paměti ap.). Pokud ve svém programu používáme výjimky (povolili jsme jejich použití ve volbách pro překladač), měli bychom ve skupině Handled by nastavit možnost User Program.

Karta Distributed Debugging obsahuje volby pro ladění distribuovaných aplikací. Toto téma přesahuje rámec naší knihy a proto se jím nebudeme zabývat.

12.5 Přizpůsobení prostředí

Nakonec se podíváme na možnosti přizpůsobení vývojového prostředí.

Nástrojové panely

Klikneme-li na nástrojový panel pravým tlačítkem myši, vyvoláme příruční nabídku, ve které můžeme určit panely, jež mají být zobrazeny, resp. skryty. Příkazem Customize... z tohoto menu vyvoláme dialogové okno, ve kterém můžeme na kartě Options mj. určit, zda se bude zobrazovat bublinová nápověda (Show Tooltips).

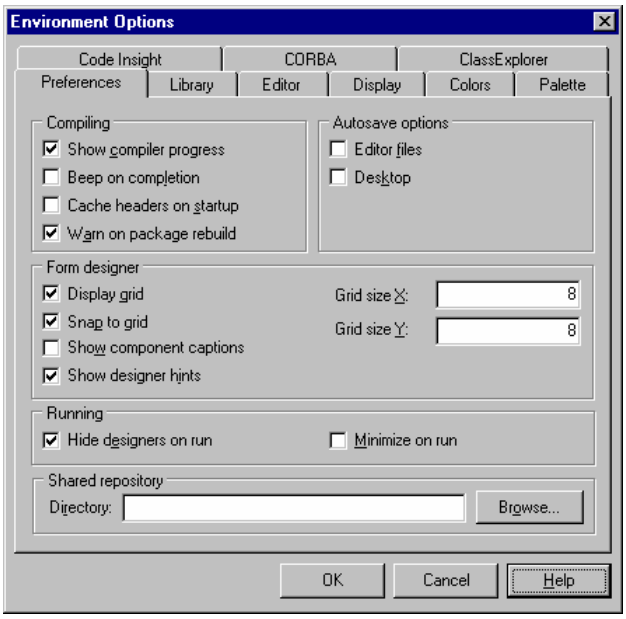
Na kartě Commands najdeme nástroje pro úpravy zobrazovaných panelů. V poli Categories najdeme jména skupin příkazů, v poli Commands příkazy a jejich tlačítka. Kterékoli z nich můžeme myší přenést na kterýkoli z panelů a naopak, kterékoli tlačítko z panelů můžeme „uklidit“ do tohoto okna.

Volby prostředí

IDE Borland C++ Builderu lze ve značné míře přizpůsobit vlastnímu vkusu a potřebám. Většinu vlastností tohoto prostředí lze nastavit pomocí dialogového okna Environment Options, které vyvoláme příkazem Tools | Environment Options... Najdeme na něm tradiční skupinu karet s volbami ovlivňujícími jednotlivé oblasti IDE.

Preferences

Na této kartě (obr. 12.26) určujeme chování IDE při překladu a běhu programu.



12.26 Preferované chování IDE

Volby ve skupině Compiling určují, zda se budou vypisovat zprávy o postupu překladu (Show compiler progress), zda má počítač po dokončení překladu pípnout (Beep on completion), zda se mají po prvním překladu uložit předkompilované hlavičkové soubory v paměti a pak používat — „kešovat“ (Cache headers on startup) a zda má prostředí vypsat varování při novém překladu a sestavování balíčku (Warn on package rebuild).

Skupina Autosave options určuje, které soubory se mají při spuštění programu v IDE nebo při uzavření C++ Builderu automaticky ukládat. Volba Editor Files nařizuje uložení všech zdrojových souborů, u kterých došlo ke změně, volba Desktop přikazuje uložit uspořádání oken na pracovní ploše v daném projektu. Při opětovném spuštění projektu se pak toto uspořádání obnoví .

Skupina Form designer upravuje chování vizuálního návrhu okna aplikace. Možnost Display Grid určuje, zda bude v návrhu okna zobrazena mřížka složená z ekvidistantních bodů, která usnadňuje zarovnávání komponent, a volba Snap To Grid určuje, zda se budou komponenty na pozice určené touto mřížkou automaticky přichycovat. Krok této mřížky ve vodorovném a svislém směru můžeme určit v polích Grid Size X a Grid Size Y. Zaškrtneme-li políčko Show Component Captions, bude se pod každou komponentou ve vizuálním návrhu zobrazovat její jméno (přesněji identifikátor ukazatele na tuto komponentu, daný vlastností Name). Poslední volba v této skupině, Show Designer Hints, určuje, zda nám

bude IDE poskytovat bublinovou nápovědu ke komponentám ve vizuálním návrhu. Pokud tuto možnost zaškrtneme a ukážeme myší na některou z komponent ve vizuálním návrhu, ukáže prostředí v bublině její typ a jméno (vlastnost Name).

Ve skupině Running najdeme volby určující chování IDE za běhu programu. Volba Minimize On Run předepisuje, že se má IDE po rozběhnutí programu minimalizovat, volba Hide Designers On Run způsobí, že se skryje okno s vizuálním návrhem, inspektor objektů atd. Zůstane okno editoru a hlavní okno IDE s nabídkou a nástrojovými panely.

Ve vstupním poli ve skupině Shared Repository můžeme předepsat adresář, ve kterém bude IDE hledat zásobník objektů (Object Repository).

Library

Na kartě Library najdeme skupinu vstupních polí, ve kterých lze zadat adresáře používané při práci s komponentami. V poli Library path je cesta ke hlavičkovým a knihovním souborům pro instalované balíčky a komponenty. Adresář uvedený v poli BPL output directory určuje místo, kam ukládat vytvořené balíčky (soubory .bpl) a adresář uvedený v poli BPI/LIB output path říká, kam se budou implicitně ukládat soubory .lib a .bpi (importní knihovny pro balíčky). Adresáře uvedené v poli Browsing path určují, kde bude prohlížeč kódu hledat zdrojové soubory, pokud je nenajde na cestách uvedených ve volbách projektu.

Podobně jako ve volbách projektu zde můžeme zadávat cesty relativně vzhledem k adresáři s instalací C++ Builderu pomocí makra $(BCB).

Editor

Jak už název karty napovídá, týkají se tyto volby chování integrovaného textového editoru.

Ve skupině Editor options najdeme dlouhou řadu voleb, které určují podrobnosti o způsobu odsazování, o chování bloků, o používání tabulátoru atd. Nebudeme je probírat všechny, zmíníme se jen o některých důležitějších.

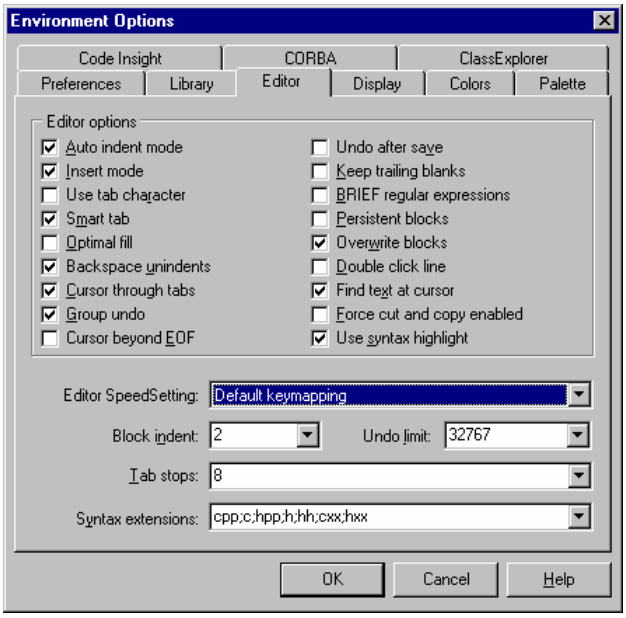
Možnost Auto Indent Mode (automatické odsazování) způsobí, že při přechodu na novou řádku umístí prostředí kurzor pod první nebílý znak na předchozí řádce. Podobně možnost Smart tab („chytrý tabulátor“) říká, že po stisknutí tabulátoru na počátku řády se kurzor umístí pod první nebílý znak na předchozí řádce.

Možnost Group undo umožňuje vracet najednou celé skupiny operací stejného druhu (např. všechna mazání znaků). Užitečná může být možnost Undo after save, která umožní vracet editovací operace i po uložení zdrojového souboru na disk.

Možnost Persistent blocks nastavuje trvalé bloky. To znamená, že blok označený např. tažením myší se pohybem kurzoru pomocí kurzorových kláves nezruší. (Toto chování bylo implicitní v dosovských překladačích, např. v Borland C++ 3.1.)

Možnost Use syntax highlight vypíná nebo zapíná zvýrazňování syntaktických kategorií ve zdrojovém textu. (Kupodivu jsou programátoři, které zvýrazňování syntaxe ruší.)

V poli Editor SpeedSeting najdeme rozbalovací seznam, hotových nastavení, mezi jinými také IDE Clasic (klasické nastavení borlandských prostředí; od implicitního se liší jen používáním persistentních bloků), emulaci Visual Studia (tj. prostředí Visual C++) atd.



12.27 Podrobnosti chování textového editoru IDE

V poli Block indent nastavujeme krok (počet pozic) pro odsazování označeného bloku jako celku (operace CTRL+K, I a CTRL+K, U). V poli Undo limit pak počet operací, které lze vrátit příkazem Edit | Undo.[65] Pole Tab stops udává interval zarážek tabulátoru a pole Syntax extensions přípony, podle kterých se bude určovat jazyk zdrojového programu při zvýrazňování syntaxe.

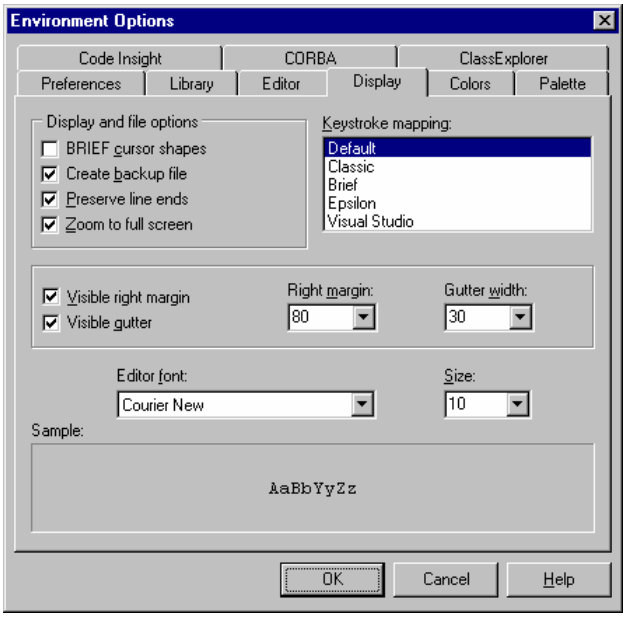
Display

Na této kartě (obr. 12.28) najdeme volby určující vzhled a chování okna editoru. Ve skupině Display and file options najdeme možnosti BRIEF cursor shapes (tvary kurzoru jako v dnes už zapomenutém editoru BRIEF, tj. textový kurzor ve tvaru podtržítka nebo obdélníčku), Create backup file (vytvářet při ukládání záložní kopie souborů), Preserve line ends (zachovávat pozice konců řádek) a Zoom to full screen (při maximalizaci vyplní okno textového editoru celou obrazovku). Poslední volba je implicitně vypnuta, okno editoru při maximalizaci vyplní jen spodní část obrazovky a ponechá viditelné hlavní okno C++ Builderu.

V seznamu Keystroke mapping najdeme různá nastavení klávesových zkratek. Implicitní nastavení, které jsme v této knize používali a popisovali, odpovídá možnosti Default. Z dalších možností uvedeme Classic (klávesové zkratky z dosovských IDE Borland C++) a Visual Studio (klávesové zkratky MS Visual C++).

V následující nepojmenované skupině voleb můžeme určit, zda se má zobrazovat pravý okraj stránky se zdrojovým kódem (Visible right margin) a levý šedý okraj (Visible gutter). Vstupní pole Right margin udává vzdálenost pravého kraje stránky od levého okraje ve znacích, pole Gutter width umožňuje nastavit šířku levého okraje okna v bodech.

[65] Větší rozsah vracení znamená také větší nároky na paměť.



Obr. 12.28 Karta Display

V seznamu Editor font můžeme předepsat písmo, které chceme v editoru používat, a v poli Size pak jeho velikost. V poli Sample uvidíme vzorek vybraného písma. Poznamenejme, že v C++ Builderu lze používat pouze neproporcionální písma (Courier, Logic ap.).

Colors

Na této kartě můžeme nastavit barvy použité pro různé syntaktické kategorie zdrojového textu, označení zarážek, aktuální pozice při krokování apod. Najdeme tu i předem připravená nastavení (pole Color SpeedSetting), z nichž asi nejdůležitější je možnost Default (implicitní nastavení). V tomto seznamu je mj. i možnost Classic, která představuje barevnou kombinaci používanou v dosovských verzích borlandských překladačů.

Poznamenejme, že nastavené barvy se ukládají do systémového registru.

Palette

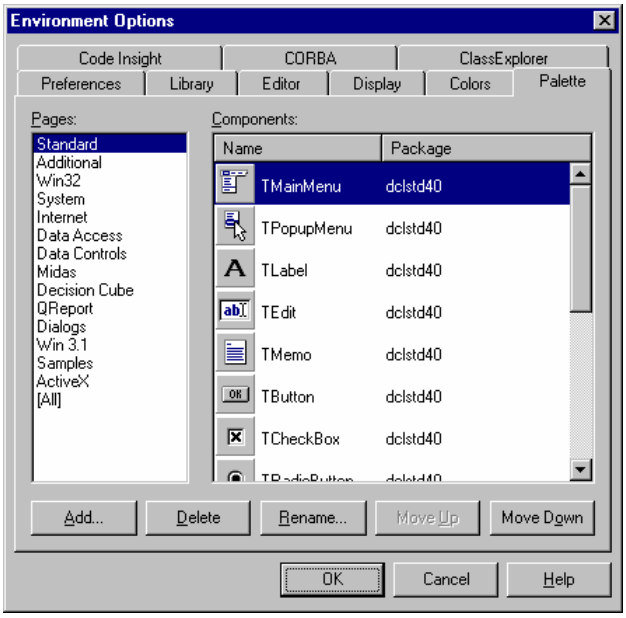
Zde můžeme přejmenovávat, přidávat nebo odebírat palety nebo komponenty na nich a měnit jejich pořadí.

V poli Pages je seznam stránek palety („seznam palet“). Chceme-li změnit jejich pořadí, můžeme je prostě přemístit myší. V poli Components je seznam komponent na právě zvolené stránce. Také jejich pořadí lze měnit přetažením myší.

Chceme-li vytvořit novou stránku, použijeme tlačítko Add. K odstranění vybrané stránky použijeme tlačítko Delete. (Odstranit lze pouze prázdnou stránku bez komponent.)

Chceme-li přemístit komponentu z jedné stránky na jinou, prostě ji vyhledáme v poli Components, přetáhneme ji myší do pole Pages a pustíme ji na jménu stránky, kam ji chceme zařadit. Komponenta se zařadí na stránku jako poslední.

Chceme-li skrýt komponentu, vybereme ji a stiskneme tlačítko Hide. Skrytá komponenta se nebude na stránkách zobrazovat. Chceme-li zobrazit skrytou komponentu, vyhledáme ji v seznamu Pages na „stránce“ All a stiskneme Show.



Obr. 12.29 Úprava palet s komponentami

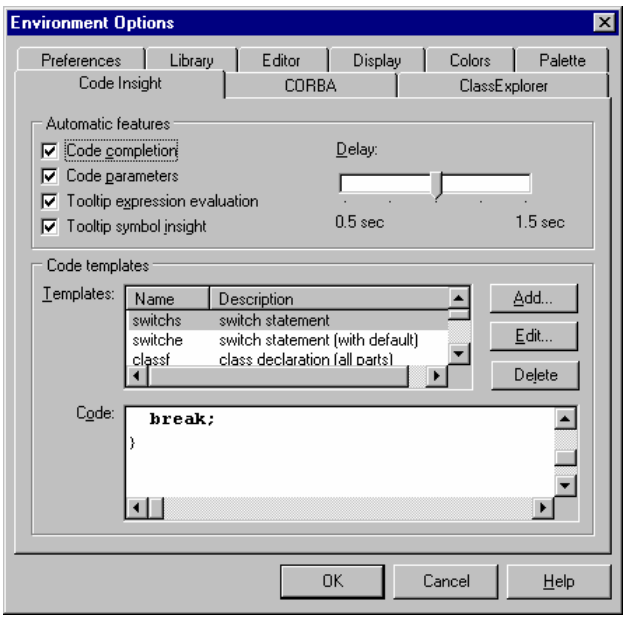
Komponenty v seznamu Components lze setřídit podle jména nebo podle balíčku stisknutím tlačítka s nápisem Name nebo Package v záhlaví tohoto seznamu.

Code Insight

Na stránce Code Insight určujeme, zda si přejeme používat některé z rafinovaných nástrojů, které s sebou přineslo IDE C++ Builderu 4. Na pomalejších počítačích totiž mohou být výrazně na obtíž.

Ve skupině Automatic features můžeme určit, zda chceme používat automatické dokončování kódu (Code completion), automatické nabízení seznamu parametrů funkcí (Code parameters), vyhodnocování výrazů v bublinové nápovědě (Tooltip expression evaluation) a bublinovou nápovědu o místě definice symbolu (Tooltip symbol insight). Pole Delay umožňuje určit prodlevu před spuštěním automatických nástrojů.

Ve skupině Code templates můžeme editovat šablony kódu pro Code Insight. Pole Templates obsahuje jednotlivé seznam šablon a jejich stručných popisů, pole Code tyto šablony ukazuje. Tlačítko Add... umožňuje přidat novou šablonu, tlačítko Delete vymazat existující šablonu.



Obr. 12.30 Volby pro Code Insight

Jednotlivé šablony můžeme editovat přímo v poli Code. Svislá čára, znak „|“, označuje pozici kurzoru po vložení šablony do zdrojového textu.

Připomeňme si, že Code Insight lze vyvolat při editování pomocí klávesové kombinace CTRL+J a nabídnutý seznam reaguje na stisknutí prvních písmen jména šablony.

Class Explorer

Tato karta obsahuje pouhé 4 volby určující chování průzkumníka tříd. Zaškrtávací pole Automatically show Explorer rozhoduje, zda se okno průzkumníka objeví po spuštění automaticky nebo zda ho budeme muset vyvolat příkazem View | ClassExplorer. Pole Show Field Types a Show Method Arguments určují, zda bude průzkumník zobrazovat u tříd typy jejich složek a zda bude zobrazovat parametry metod. Pole Show Warnings určuje, zda bude průzkumník vypisovat varování při chybách, které nejsou kritické (není dostupný soubor ap.).

CORBA

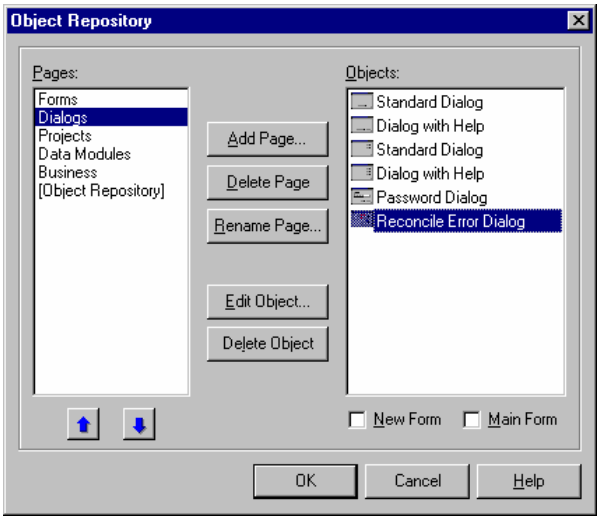
Zde najdeme volby, které se týkají distribuovaných aplikací. Toto téma přesahuje rámec naší knihy, a proto se jimi nebudeme zabývat.

12.6 Zásobník objektů

Zásobník objektů (Object repository) vyvoláme příkazem File | New. Najdeme v něm několik karet s ikonami šamanů, kteří umožňují vytvořit novou aplikaci různých typů, nové okno, databázovou tabulku, dialogové okno atd.

Do zásobníku můžeme také přidat své vlastní objekty, to jsme viděli v kap. 4.

Chceme-li zásobník objektů upravit, použijeme příkaz Tools | Object repository.... Tím vyvoláme dialogové okno, které vidíte na obr. 12.31.



Obr. 12.31 Upravujeme zásobník objektů

V poli Pages vidíme seznam karet zásobníku; chybí první, protože tu nelze upravovat a nelze na ni ani přidat nový objekt nebo některý z objektů odstranit. V poli Objects je seznam objektů uložených na dané stránce.

Tlačítka se šipkami umožňují měnit pořadí karet. Tlačítka Add Page..., Delete Page a Rename Page... umožňují přidat novou kartu, odstranit nebo přejmenovat zvolenou kartu. Tlačítko Edit Object... vyvolá dialogové okno, ve kterém můžeme změnit popis objektu, a tlačítko Delete Object umožňuje vybraný objekt ze zásobníku odstranit.