1. Seznámení s C++ Builderem

1. Po spuštění C++ Builderu jsou na obrazovce zobrazena tato okna: okno Builderu (je umístěno u horního okraje obrazovky), okno Inspektora objektů (je v levé části obrazovky), okno návrhového formuláře a okno Editoru kódu formuláře (poslední z těchto oken je překryto předposledním). Okno Inspektora objektů má dvě stránky: stránku **Properties** (vlastností) a stránku **Events** (událostí). Nejprve se budeme zabývat stránkou vlastností. Jsou zde uvedeny různé vlastnosti formuláře. Mezi důležité vlastnosti formuláře patří vlastnosti **Height** (výška) a **Width** (šířka). Změňte pomocí myši rozměry okna formuláře na obrazovce a sledujte, jak se tyto změny projevují na odpovídajících vlastnostech v okně Inspektora objektů. Lze to provést i obráceně. Když změníme některou z těchto vlastností v Inspektoru objektů, změní se i rozměr formuláře na obrazovce.
2. Další důležité vlastnosti určují pozici formuláře na obrazovce. Jsou to vlastnosti **Left** (vzdálenost od levého okraje obrazovky) a **Top** (vzdálenost od horního okraje obrazovky). Přesvědčte se, že to funguje stejným způsobem jako v bodě 1.
3. Vlastnost **Caption** (název okna) určuje text zobrazený v horním řádku formuláře (okna). Změňte tento název na „Normální okno“.
4. Vlastnost **Color** (barva formuláře) se určuje pomocí symbolických konstant. Zjistěte, jaká jména mají tyto konstanty.
5. Nejdůležitější vlastností je vlastnost **Name** (jméno). Pomocí této vlastnosti se odkazujeme na formulář, případně na jiné objekty umístěné na formuláři. Hodnota této vlastnosti musí být identifikátor. Tzn. nelze zde např. používat mezery, písmena s diakritikou. Hodnotu této vlastnosti vytváří Builder, ale můžeme ji změnit.
6. Vytvořenou aplikaci můžeme kdykoliv spustit a prohlédnout si tak výsledky své práce. Provedeme to volbou **Run | Run**. Po provedení překladu (trvá to několik sekund) se na obrazovce zobrazí náš formulář jako okno (zatím prázdné), se kterým můžeme běžným způsobem manipulovat (přesouvat, měnit jeho velikost apod.). Uzavřením okna se opět dostaneme do vývojového prostředí Builderu.
7. Na formulář můžeme vkládat různé komponenty. Používáme k tomu Paletu komponent, která je umístěna v pravé dolní části okna Builderu (má několik stránek: **Standard**, **Additional**, atd.). Paleta komponent je tvořena řadou tlačítek, ukázáním myší na tlačítko zobrazíme název tlačítka. Nalezněte tlačítko s názvem **Button** (je na stránce **Standard**).
8. Komponentu na formulář vložíme tak, že kliknutím vybereme komponentu na Paletě komponent a dalším kliknutím ji umístíme na požadované místo formuláře. Vložte na formulář komponentu **Button (**tlačítko).
9. Jestliže vložená komponenta je vybrána (je obklopena záchytnými čtverečky), pak můžeme měnit její velikost a umístění a to stejným způsobem, jako jsme to měnili u formuláře. Vlastnosti **Top** a **Left** se nyní vztahují k formuláři. Vlastnost **Caption** komponenty tlačítka určuje text zobrazený na tlačítku. U našeho tlačítka změňte tento text na „Zelený“.
10. Dále zapíšeme příkazy, které se mají provést při stisknutí tlačítka. Dvojitým kliknutím na tlačítku vytvoříme kostru obsluhy události stisknutí tlačítka (jedná se o funkci, přičemž Builder vytvoří její hlavičku a složené závorky pro zápis těla funkce) a tato (zatím prázdná) funkce je zobrazena v Editoru kódu s kurzorem umístěným mezi { }. Do tohoto místa budeme zapisovat příkazy, které se mají provést při stisknutí tlačítka. V našem případě sem zapíšeme příkazy:
    ```
    Color = clGreen;
    Caption = "Zelené okno";
    ```
    První příkaz změní barvu formuláře na zelenou a druhý změní název formuláře (změníme vlastnosti **Color** a **Caption** formuláře). Když nyní aplikaci přeložíme a spustíme, pak po stisknutí našeho tlačítka tyto akce proběhnou.
11. Vlastnosti objektů tedy můžeme měnit pomocí Inspektora objektů (při návrhu), nebo příkazy jazyka C++ (za běhu aplikace). Jestliže v programu zapíšeme některý příkaz chybně je signalizována chyba. Udělejte v zapsaných příkazech nějakou chybu (např. vynechejte některý středník nebo zkomolte některý identifikátor) a pokuste se aplikaci znovu spustit. Podívejte se na způsob signalizace chyby. Chybu opět odstraňte.
12. Přidejte na formulář další tlačítka, např. tlačítka „Červený“, „Modrý“ atd. a zajistěte, aby při stisknutí z těchto tlačítek se odpovídajícím způsobem změnila barva a název formuláře. Program vyzkoušejte.
13. Na formulář přidáme ještě tlačítko, kterým budeme moci formulář uzavřít (ukončit běh programu). Toto tlačítko nazveme např. „Konec“ a obsluha stisku tohoto tlačítka bude tvořena příkazem:
    ```
    Close();
    ```
14. V okně Editoru kódu si můžeme prohlednout vytvořený program (přesněji řečeno programovou jednotku formuláře). Každý formulář má svoji programovou jednotku. V tomto zdrojovém souboru jsou uvedeny námi vytvořené obsluhy událostí pro ošetření stisknutí našich tlačítek. Jestliže programovou jednotku

formuláře uložíme do souboru, pak Builder vyžaduje použití přípony CPP. Každý formulář má mimo své programové jednotky další soubor, obsahující popis vzhledu formuláře (vlastnosti formuláře a objektů umístěných na formuláři). Tento soubor má příponu DFM, jedná se o binární soubor, který si běžným způsobem nemůžeme prohlédnout (není to ani potřeba, neboť vzhled formuláře vidíme). Jméno tohoto souboru je stejné jako jméno programové jednotky formuláře (liší se pouze příponou). Implicitní jméno souborů prvního formuláře je *Unit1* (toto jméno můžeme při ukládání souborů změnit). Existuje ještě hlavičkový soubor formuláře. Jeho implicitní jméno je *Unit1.H*.

15. Projekt obsahuje mimo souboru programové jednotky formuláře, hlavičkového souboru formuláře a souboru formuláře (těchto souborů může být i více, skládá-li se projekt z více formulářů) i soubory projektu. Každý projekt má jeden projektový soubor, který je vytvářen a udržován Builderem a my bychom jej neměli měnit. Tento soubor má příponu CPP. Implicitní jméno je *Project1*. *Je vhodné každý projekt ukládat do samostatného adresáře.* Soubory projektu a formuláře uložíme příkazem **File | Save All**. Po zadání příkazu se nás Builder ptá na jména programových jednotek a jméno projektového souboru. Vytvořený projekt uložte do adresáře C:\TEMP\PR1 (pro soubory použijte implicitní jména). Po uložení projektu jej znovu přeložte a vyzkoušejte. Podívejte se také do adresáře C:\TEMP\PR1 jaké obsahuje soubory. Je zde také soubor s příponou EXE. Tento soubor obsahuje náš přeložený projekt. Můžeme jej spustit přímo ve Windows (není nutno jej spouštět z prostředí Builderu).
16. Dále začneme vytvářet nový projekt. K otevření nového projektu použijeme příkaz **File | New Application**. Tím vytvoříme prázdnou aplikaci (jako při spuštění Builderu). Do středu formuláře vložíme komponentu **Edit**. Když nyní spustíme naší aplikaci můžeme do této komponenty zapisovat libovolný text. Původní text zobrazený v této komponentě byl „Edit1“, je to hodnota vlastnosti **Text** editačního ovladače. Zrušíme hodnotu této vlastnosti a v ovladači nebude nic zobrazeno.
17. V naší aplikaci budeme zjišťovat počet znaků zapsaného textu. Potřebujeme tedy vypisovat výsledek. K výpisu použijeme komponentu **Label**. Umístíme tuto komponentu na formulář pod editační komponentu. Implicitní text je „Label1“, je to hodnota vlastnosti **Caption**. Zrušíme tento text (po zrušení textu komponenta **Label** nebude vidět). Název formuláře změníme na „Zjišťování délky textu“.
18. Na formulář přidáme ještě tlačítko, jehož stisknutí spustí proces zjištění délky zapsaného textu. Tlačítko nazveme „Zjisti“. Dále musíme vytvořit obsluhu události stisknutí tlačítka. V této obsluze zjistíme délku textu metodou **Length**, kterou aplikujeme na instanci vlastnosti **Text** editačního ovladače **Edit1**). Tím získáme číselnou hodnotu (číselnou hodnotu nelze přímo zobrazit; hodnota vlastnosti **Caption** je typu **AnsiString**; je to třída), kterou musíme převést na řetězec znaků konstruktorem **AnsiString**. Aby se výsledek zobrazil, je třeba vytvořený řetězec přiřadit do vlastnosti **Caption** komponenty **Label1**. Obsluha bude tedy tvořena příkazem:
    ```
    Label1->Caption = "Počet znaků = " + AnsiString(Edit1->Text.Length());
    ```
    Jestliže pracujeme s vlastností nějaké komponenty, pak zapisujeme jméno komponenty, -> a jméno vlastnosti, když pracujeme s vlastností samotného formuláře, pak zapisujeme pouze jméno vlastnosti (i v tomto případě bychom mohli použít jméno formuláře, např. Form1->Color).
19. Aplikaci můžeme vyzkoušet. Bylo by asi výhodné, aby při stisknutí klávesy Enter výpočet délky proběhl automaticky, abychom nemuseli „mačkat“ tlačítko. Zde využijeme to, že formulář může obsahovat jedno implicitní tlačítko, které se stiskne při stisku klávesy Enter. Implicitní tlačítko vytvoříme nastavením vlastnosti tlačítka **Default** na **True** (změnu hodnoty **False** na **True** a naopak v Inspektoru objektů dosáhneme dvojitým kliknutím na hodnotě). Proveďte toto vylepšení naší aplikace.
20. Nyní se pokusíme tuto aplikaci změnit a to tak, že zobrazená informace o délce textu bude při zapisování textu stále aktualizována. V naší aplikaci tedy zrušíme tlačítko (a obsluhu jejího stisku) a využijeme to, že při zápisu každého znaku (změně textu v editačním ovladači) vzniká událost **OnChange**. Tato událost je implicitní událostí editačního ovladače a kostru její obsluhy můžeme vytvořit dvojitým kliknutím na editačním ovladači. Obsluha této události bude tvořena stejným příkazem jako v předchozí verzi byla obsluha stisku tlačítka. Aplikaci spustíme a vidíme, že při zápisu textu do editačního ovladače, se hodnota délky textu stále mění. Zajistěte ještě, aby tato informace byla zobrazena i před zahájením zápisu.
21. Začneme opět s novou aplikací. Na formulář umístíme editační ovladač, komponentu **ListBox** (okno seznamu řetězců) a dvě tlačítka. Formulář nazveme „Demonstrace seznamu“, editační ovladač vyprázdníme a tlačítka popíšeme „&Přidat“ a „&Vyprázdnit“ (znaky & používáme v popisu tlačítek pro označení zkracovacích kláves). Tlačítko „Přidat“ nastavíme jako implicitní. Při stisku tlačítka „Přidat“ vezmeme text zapsaný v editačním ovladači a přidáme jej do seznamu řetězců. Vytvoříme tedy obsluhu stisknutí tohoto tlačítka s příkazem:
    ```
    ListBox1->Items->Add(Edit1->Text);
    ```

Zde jsme použili metodu **Add** přidávající parametr do vlastnosti **Items** (seznam řetězců) komponenty **ListBox1**. Obdobně vytvoříme obsluhu stisknutí druhého tlačítka (použijeme zde metodu **Clear** vlastnosti **Items**; je bez parametrů).

22. Když naši aplikaci vyzkoušíme, zjistíme několik nedostatků. Bylo by vhodné, aby po přidání textu do seznamu byl text v editačním ovladači zrušen. Dále by bylo vhodné, aby po vložení textu do editačního ovladače jsme mohli ihned začít psát do editačního ovladače další text. Do obsluhy události stisknutí tlačítka „Přidat“ vložíme příkazy:
```
Edit1->Text = "";
Edit1->SetFocus();
```
23. Nyní již naše aplikace pracuje uspokojivě. Můžeme ale přijít s požadavkem, aby seznam řetězců byl uspořádán abecedně. Tento problém můžeme vyřešit pomocí vlastnosti **Sorted** seznamu řetězců. Pokuste se provést tuto změnu.
24. Když se podíváme na některou obsluhu události (např. obsluhu události stisku tlačítka „Přidat“), pak zjistíme, že jméno obsluhy je tvořeno jménem formuláře, dvěmi dvojtečkami a jménem tvořeným z názvu komponenty a názvu události (např. Form1::Button1Click). Zatím jsme používali převážně obsluhy stisknutí tlačítka. Události je ale mnohem více. Jestliže chceme vytvořit obsluhu pro nějakou událost (stisknutí tlačítka je implicitní událostí tlačítka a obsluhu implicitní události vytvoříme dvojitým kliknutím na komponentě) provedeme to dvojitým kliknutím na události v Inspektoru objektů. Začneme vytvářet novou aplikaci, kde do středu formuláře vložíme tlačítko a vytvoříme obsluhu události **OnResize** formuláře tak, aby při změně rozměrů formuláře myší, bylo tlačítko vždy uprostřed. Rozměry vnitřní oblasti formuláře určují vlastnosti **ClientWidth** a **ClientHeight**. Vytvořte tuto aplikaci sami.

2. Procvičování C++

1. Vytvoříme aplikaci, která bude převádět sekundy na minuty a sekundy. Na formulář umístíme editační komponentu (vyprázdníme její obsah), nad ní umístíme komponentu **Label** s textem „Zadej počet sekund:“, vedle editační komponenty přidáme tlačítko „Vypočti“ a pod editační komponentu vložíme další komponentu **Label**, ve které budeme zobrazovat výsledek (tuto komponentu vyprázdníme). Přiřazovací příkazy v jazyku C++ používáme běžným způsobem. Musíme dbát na kompatibilitu přiřazované hodnoty s typem, do kterého přiřazujeme. Pro převod řetězce na celé číslo používáme metodu **ToInt** a pro opačný převod konstruktor **AnsiString** (vlastnosti Text a Caption jsou typu AnsiString). Obsluhu stisku našeho tlačítka bude tvořit příkaz:
```
Label2->Caption = "Výsledek: " + AnsiString(Edit1->Text.ToInt() / 60)
                    + ":" + AnsiString(Edit1->Text.ToInt() % 60);
```
   V našem programu není ošetřeno případné zadání nečíselné hodnoty (je pouze generována výjimka popisující chybu).
2. Upravte předchozí program tak, aby se počet sekund rozložil na počet hodin, minut a sekund. Pokud budete potřebovat pomocnou proměnnou na uložení mezivýsledku, můžete deklarovat lokální proměnnou v obsluze obvyklým způsobem.
3. Vytvořte program, který bude provádět výpočet obvodu a obsahu kružnice o zadaném poloměru (každý výsledek uložte do jiné komponenty **Label**). Pro uložení reálných čísel používáme typy **float**, **double** nebo **long double** (základní z těchto typů je typ **double**). Pro převod řetězce na typ **double** používáme metodu **ToDouble** a pro opačný převod opět konstruktor **AnsiString**.
4. V následujícím programu (začneme novou aplikaci) budeme název okna určovat v závislosti na zadané hodnotě (pro hodnoty větší než 100 bude název „Velké číslo“, jinak „Malé číslo“). Na formulář vložíme editační ovladač (vyprázdníme jej), nad něj umístíme **Label** s textem „Zadej celé číslo:“ a vedle něj tlačítko „Vyhodnoť“. Obsluhu stisku tohoto tlačítka bude tvořit příkaz:
```
if (Edit1->Text.ToInt() > 100) Caption = "Velké číslo";
else Caption = "Malé číslo";
```
   Vidíme, že příkaz **if** se používá běžným způsobem.
5. Vytvořte program s formulářem obsahujícím dva editační ovladače, ve kterých budete zadávat celá čísla a tlačítko, po jehož stisku se větší číslo zobrazí jako název okna. Editační ovladače doplňte vhodným textem.
6. Začneme novou aplikaci. V komponentě **Memo** můžeme zobrazit obsah textového souboru. Vytvoříme formulář, na který vložíme tuto komponentu (je na stránce **Standard**) a zvětšíme ji tak, aby zabírala levou polovinu formuláře. Když nyní aplikaci spustíme, můžeme komponentu **Memo** používat jako textový editor. Vyzkoušejte. K otevření textového souboru v této komponentě použijeme metodu *LoadFromFile* pro vlastnost **Lines** této komponenty. Parametrem metody je specifikace otevíraného souboru. Většinou budeme chtít

specifikovat soubor až za běhu aplikace a využijeme tedy komponentu **OpenDialog** (je na stránce **Dialogs**). Tuto komponentu umístíme na formulář (za běhu aplikace nebude viditelná) a na formulář dále vložíme tlačítko a změníme jeho text na „Soubor“. Vytvoříme obsluhu stisku tohoto tlačítka příkazem:

```
if (OpenDialog1->Execute()) Memo1->Lines->LoadFromFile(OpenDialog1->FileName);
```

Když nyní aplikaci spustíme a stiskneme tlačítko „Soubor“ je otevřeno dialogové okno pro otevření souboru. Zavřeme-li toto okno bez volby jména souboru (např. okno uzavřeme tlačítkem **Zrušit**), pak funkce *Execute* (zobrazující toto okno) vrací *False* a příkaz v **if** se neprovede. Vybereme-li jméno textového souboru, pak obsah tohoto souboru je zobrazen v komponentě **Memo**. Při delších souborech by bylo vhodné komponentu **Memo** vybavit posuvníky. To můžeme zajistit změnou vlastnosti **ScrollBars** této komponenty. Přidejte do komponenty oba posuvníky. Komponenta **OpenDialog** má vlastnost **Filter**. Pomocí této vlastnosti můžeme určovat typy souborů, které budou nabízeny k otevření. V poli hodnot této vlastnosti stiskneme tlačítko (…) a v zobrazeném Editoru filtrů zadáme požadovaný filtr. V prvním sloupci zadáme název filtru (text zobrazovaný v dialogovém okně otevření souboru) a ve druhém sloupci tento filtr vyjádříme pomocí hvězdičkové konvence.

7. Vytvořený textový editor můžeme také vybavit funkcemi pro práci se schránkou a funkcí zrušení celého textu. Přidáme tedy na formulář tlačítka „Vyjmout“, „Kopírovat“, „Vložit“ a „Smazat vše“. Obsluha pro stisknutí tlačítka „Vyjmout“ bude tvořena příkazem:

   ```
   Memo1->CutToClipboard();
   ```

   Obdobně v dalších obsluhách událostí použijeme metody *CopyToClipboard*, *PasteFromClipboard* a *Clear*. Vytvořte tyto obsluhy a program vyzkoušejte.
8. Po provedení změn v textu souboru budeme chtít soubor většinou uložit. Můžeme postupovat obdobně jako při otevírání souboru, použijeme komponentu **SaveDialog** a metodu *SaveToFile* (na formulář přidáme tlačítko „Ulož“).
9. Začneme novou aplikaci, kde se budeme zabývat změnou barvy. Pro změnu barvy běžně používáme standardní komponentu **ColorDialog** (používá se obdobně jako **SaveDialog**). Tzn. dialogové okno barev zobrazíme metodou *Execute* a vybraná barva je určena vlastností **Color** této komponenty. Vytvořte aplikaci, kde po stisku tlačítka „Barva“ zobrazíte dialogové okno barev a vybranou barvu přiřadíte barvě formuláře.
10. V následující nové aplikaci na formulář přidáme editační ovladač (vyprázdníme jej), nad něj přidáme komponentu **Label** s textem „Zadej číslo dne v týdnu:“, vedle editačního ovladače umístíme tlačítko „Urči“ a pod editační ovladač vložíme další komponentu **Label** pro výpis výsledků (vyprázdníme jej). Výsledky budeme zobrazovat větším písmem a barevně. Nastavíme tedy vlastnost **Font** komponenty **Label2** (po výběru vlastnosti **Font** v Inspektoru objektů se v místě hodnoty této vlastnosti zobrazí tlačítko s třemi tečkami; kliknutím na toto tlačítko zobrazíme dialogové okno volby písma, kde provedeme požadované nastavení) na větší a barevné písmo. Obsluha stisknutí tlačítka bude tvořena příkazem:

    ```
    switch (Edit1->Text.ToInt()) {
    case 1: Label2->Caption = "Pondělí"; break;
    case 2: Label2->Caption = "Úterý"; break;
    case 3: Label2->Caption = "Středa"; break;
    …
    default: Label2->Caption = "Chyba";
    }
    ```

    Chybějící příkazy doplňte sami. Vidíme, že příkaz **switch** se používá běžným způsobem.
11. Vytvoříme nový prázdný projekt a umístíme na formulář seznam řetězců (**ListBox** – zvětšíme jeho velikost a změníme v něm typ písma na *Courier New*) a dvě tlačítka (s texty **For** a **While**). Obsluha stisku prvního tlačítka bude tvořena příkazy:

    ```
    char pom[30];
    ListBox1->Items->Clear();
    for (int i = 1; i <= 20; i++){
      sprintf(pom, "Řetězec %3d", i);
      ListBox1->Items->Add(pom);
    }
    ```

    Pro druhé tlačítko vytvoříme tuto obsluhu:

    ```
    char pom[30];
    ListBox1->Items->Clear();
    randomize();
    int i = 0;
    while (i < 1000) {
      i = i + random(100);
      sprintf(pom, "Náhodné číslo:%5d", i);
    ```

```
  ListBox1->Items->Add(pom);
}
```

Tato aplikace ukazuje použití cyklů a generování náhodných čísel. Zjistěte co provádí.

12. Vytvořte aplikaci, ve které budete zobrazovat tabulku funkčních hodnot funkcí sinus a cosinus od 0 do 90 stupňů s krokem 10. Na formulář umístěte komponentu **ListBox** a zvětšíme ji. Tabulku vypisujte ve dvou sloupcích; první sloupec nadepište „Úhel“ a druhý „Sinus“, resp. „Cosinus“. Na formulář ještě vložíme tlačítka „Sinus“ a „Cosinus“. Vytvořte obsluhy stisku tlačítek tak, aby vždy při stisknutí některého tlačítka byly zobrazeny funkční hodnoty příslušné funkce.
13. Vzhlednější řešení předchozího zadání můžeme získat pomocí komponenty **StringGrid**. S používáním této komponenty se seznámíme v nové aplikaci. Na formulář umístíme tuto komponentu a tlačítko „Spusť“. Formulář zvětšíme. U komponenty **StringGrid** změníme dále tyto vlastnosti: **ColCount** (počet sloupců) nastavíme na 6, **RowCount** (počet řádků) na 5, **DefaultColWidth** (implicitní šířka sloupce) na 100 a **ScrollBars** na *ssNone* (zákaz použití posuvníků). Komponentu zvětšíme tak, aby všechny sloupce a řádky byly zobrazeny. Obsluha stisku tlačítka bude tvořena příkazy:

```
char pom[20];
int Sloupec, Radek;
for (Sloupec = 1; Sloupec <= 5; Sloupec++) {
  sprintf(pom, "Sl. %d", Sloupec);
  StringGrid1->Cells[Sloupec][0] = pom;
}
for (Radek = 1; Radek <= 4; Radek++) {
  sprintf(pom, "Rad. %d", Radek);
  StringGrid1->Cells[0][Radek] = pom;
}
for (Sloupec = 1; Sloupec <= 5; Sloupec++)
  for (Radek = 1; Radek <= 4; Radek++) {
    sprintf(pom, "Sl. %d Rad. %d", Sloupec, Radek);
    StringGrid1->Cells[Sloupec][Radek] = pom;
}
```

Prostudujte si výstup a zjistěte, jak se používá vlastnost **Cells** komponenty **StringGrid**. Zjistěte k čemu se dají použít vlastnosti **FixedColor**, **FixedCols** a **FixedRows**.

14. Zadání 12 vyřešte pomocí komponenty **StringGrid**. Vytvořte tabulku o 2 sloupcích a 11 řádcích. Hodnoty prvního sloupce jsou stále stejné. Můžeme je tedy zobrazit v obsluze události **OnCreate** formuláře.
15. Vytvoříme aplikaci, kde na formuláři budou tři editační ovladače (vyprázdníme je a před ně umístíme texty „První operand:“, „Druhý operand:“ a „Výsledek:“) a tlačítka „Součet“, „Rozdíl“, „Součin“ a „Podíl“. Obsluha události stisknutí tlačítka „Součet“ bude tvořena příkazy:

```
int x, y;
x = Edit1->Text.ToInt();
y = Edit2->Text.ToInt();
Edit3->Text = AnsiString(x + y);
```

Vytvořte obsluhy i pro ostatní tlačítka a program vyzkoušejte.

16. Do předchozí aplikace přidáme ještě další editační ovladač, ve kterém budeme počítat počet provedených výpočtů (vyprázdníme jej a před něj umístíme text „Počet provedených výpočtů:“). Pro počítání provedených výpočtů musíme zavést globální proměnou, nazveme ji např. *Pocet* (globální proměnné deklarujeme mimo funkce; je nutno ji deklarovat před první funkcí, ve které ji používáme). Do všech obsluh stisknutí tlačítek přidáme příkazy:

```
Pocet++;
Edit4->Text = AnsiString(Pocet);
```

Bylo by vhodné naši proměnnou *Pocet* na počátku výpočtu vynulovat. Můžeme to provést tak, že její deklaraci spojíme s inicializací, tzn. před první obsluhu vložíme:

```
int Pocet = 0;
```

17. Jistě jste si povšimli, že když stisknete některé tlačítko a nemáte zadané operandy je signalizována chyba. Na začátku obsluh stisku tlačítek budeme testovat, zda oba operandy jsou zadány. Toto testování je nutno provádět ve všech obsluhách stisku tlačítek a tedy vytvoříme funkci, která otestuje editační ovladač zadaný parametrem funkce (pokud editační ovladač je prázdný vrátí *True*, jinak vrátí *False*). Tato funkce bude vypadat takto:

```
bool NeniHodnota(TEdit *EditOvl){
  if (EditOvl->Text == "") {
    EditOvl->Color = clRed;
    EditOvl->Text = "Zadej hodnotu";
```

```
    EditOvl->SetFocus();
    return true;
  }
  else {
    EditOvl->Color = clWindow;
    return false;
  }
}
```

Parametr funkce je typu ukazatel na *TEdit*, což je typ editačního ovladače (všechny typy komponent začínají písmenem T) a jako skutečný parametr tedy budeme moci použít *Edit1* i *Edit2*. Pokud v editačním ovladači není zapsán žádný text, je změněna barva ovladače na červenou a je zde vypsán text „Zadej hodnotu". Příkaz `EditOvl->SetFocus();` zajistí vybrání testovaného editačního ovladače a můžeme tedy do něj ihned zapisovat hodnotu. Pokud editační ovladač není prázdný, pak je změněna jeho barva na normální barvu okna (*clWindow*). Tuto funkci musíme ještě vyvolat na začátku všech obsluh stisků tlačítek. Na začátek všech obsluh tedy přidáme příkaz:

```
if (NeniHodnota(Edit1) || NeniHodnota(Edit2)) return;
```

Proveďte výše uvedené změny pro všechna tlačítka a vyzkoušejte.

18. V další aplikaci se seznámíme s používáním komponenty **Timer** (časovač - je ji možno použít k definování časového intervalu). Na formulář umístíme tuto komponentu (je na stránce **System** Palety komponent). Jedna z vlastností této komponenty určuje časový interval v milisekundách (tato vlastnost se jmenuje **Interval**). Do programu vložíme deklaraci globální proměnné **X** typu **bool** a vytvoříme obsluhu události **OnTimer** časovače s příkazy:

```
if (X) Color = clRed;
else Color = clGreen;
X = !X;
```

Zjistěte, co provádí tato aplikace.

19. Vytvořte aplikaci, ve které se pokusíte simulovat házení hrací kostkou. Každou sekundu generujte hod a jeho výsledek zobrazte na formuláři (číselně). Inicializaci generátoru náhodných čísel proveďte v obsluze události **OnCreate** formuláře.

3. Používání komponent

1. Nejprve se začneme zabývat formulářem. Mimo vlastností, se kterými jsme se již seznámili, má formulář další důležité vlastnosti. Vlastnost **BorderStyle** určuje styl okraje formuláře (např. hodnota *bsNone* určuje formulář bez okrajů, tzn. nelze měnit jeho velikost). Měňte tuto vlastnost a vyzkoušejte, jak se změna projeví na vzhledu a chování formuláře za běhu aplikace.
2. Formulář má dále vlastnost **BorderIcons** (ikony rámu formuláře, tj. minimalizační a maximalizační tlačítka apod.). Před jménem této vlastnosti je v Inspektoru objektů uveden znak + (indikace, že vlastnost se skládá z podvlastností). Klikneme-li dvojitě na takto označeném jméně vlastnosti, jsou na následujících řádcích zobrazeny podvlastnosti a znak + se změní na - (opětovným dvojitým kliknutím se vrátíme k původnímu zobrazení). Při tomto „rozbalení" vlastnosti, můžeme jednotlivé podvlastnosti nastavovat samostatně. Pokuste se z formuláře odstranit maximalizační tlačítko (**BorderStyle** musí být nastaveno na *bsSizeable*; změna se projeví až při spuštění aplikace).
3. Vlastnosti **Position** a **WindowState** určují způsob zobrazení formuláře na obrazovce. **Position** určuje, zda formulář získá pozici a velikost danou při návrhu, nebo zda se použije velikost a pozice navržená Windows za běhu aplikace. **WindowState** určuje počáteční stav formuláře (minimalizovaný, maximalizovaný nebo normální). Pokuste se změnit některou z těchto vlastností a vyzkoušejte jak se změna projeví.
4. Vlastnost **ShowHint** komponenty určuje, zda komponenta zobrazí nápovědu, když se na ní na okamžik zastavíme kurzorem myši. Zobrazený text je určen vlastností **Hint**. Umístěte na formulář nějakou komponentu a zajistěte pro ní zobrazování nápovědy.
5. Komponentu **Panel** lze používat pro ukládání dalších komponent. Můžeme z ní vytvořit např. stavový řádek nebo na ní vložit komponenty, které tvoří nějaký logický celek. Vlastnost **Align** určuje umístění komponenty. Možné hodnoty této vlastnosti mají následující význam: *alTop* (panel je umístěn na horní okraj formuláře, zabírá celou šířku formuláře a sleduje i změnu šířky formuláře; obdobně platí i pro další hodnoty této vlastnosti), *alRight* (panel je umístěn na pravý okraj formuláře), *alBottom* (spodní okraj), *alLeft* (levý okraj), *alClient* (celá plocha formuláře) a *alNone* (nemění se; zůstává tak jak nastavil uživatel). Umístěte na formulář komponentu panelu a vyzkoušejte vliv hodnot vlastnosti **Align** na umístění komponenty.

6. U komponenty **Panel** lze pomocí vlastností **BevelInner**, **BevelOuter**, **BevelWidth**, **BorderStyle** a **BorderWidth** měnit způsob orámování panelu. Nastavte hodnotu vlastnosti **BevelWidth** na 10 (pro lepší viditelnost) a vyzkoušejte vliv změn ostatních těchto vlastností na zobrazení okraje.
7. Další komponentou, se kterou se seznámíme, je komponenta **Shape**. Tato komponenta umožňuje na formulář vkládat základní geometrické tvary. Vložený tvar je určen vlastností **Shape**, která v rozbalovacím seznamu nabízí možné hodnoty (např. *sstCircle* pro kružnici). Zjistěte, jak se zobrazí další nabízené hodnoty. Zobrazený geometrický tvar ovlivňují také vlastnosti **Pen** (ovlivňuje okraj tvaru; barva, šířka a styl čáry) a **Brush** (ovlivňuje vnitřek tvaru; barva a výplňový vzor). Vyzkoušejte.
8. Začneme vytvářet novou aplikaci a to signalizaci na železničním přejezdu. Doprostřed formuláře umístíme komponentu **Panel**, zrušíme její titulek a vložíme na ní vedle sebe dvě komponenty **Shape**. Tyto komponenty změníme na kružnice. Na formulář déle vložíme komponentu **Timer** a zapíšeme pro ní následující obsluhu události **OnTimer**.
```
if (Shape1->Brush->Color != clRed){
  Shape1->Brush->Color = clRed;
  Shape2->Brush->Color = clWhite;
}
else {
  Shape1->Brush->Color = clWhite;
  Shape2->Brush->Color = clRed;
}
```
9. Naši aplikaci vylepšíme přidáním tlačítka, jehož stisknutí mění režim signalizace (blikání, neblikání). Na formulář přidáme tlačítko s textem „Vlak projel“, přidáme globální proměnnou typu **bool** indikující stav vlaku (**true** = vlak jede), nastavíme počáteční hodnotu této proměnné na **true**, vytvoříme událost obsluhující stisk tlačítka a změníme obsluhu události časovače:
```
bool Vlak = true;
void __fastcall TForm1::Timer1Timer(TObject *Sender)
{
  if (Vlak){
    if (Shape1->Brush->Color != clRed){
      Shape1->Brush->Color = clRed;
      Shape2->Brush->Color = clWhite;
    }
    else {
      Shape1->Brush->Color = clWhite;
      Shape2->Brush->Color = clRed;
    }
  }
  else {
    Shape1->Brush->Color = clWhite;
    Shape2->Brush->Color = clWhite;
  }
}
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  if (Vlak) Button1->Caption = "Vlak se blíží";
  else Button1->Caption = "Vlak projel";
  Vlak = !Vlak;
}
```
10. Přidejte do aplikace třetí (fialové) blikající světlo signalizující bezpečný přejezd.
11. Pokuste se vytvořit aplikaci znázorňující semafor na křižovatce. K dosažení různého časového intervalu pro zobrazení červené (nebo zelené) a oranžové používejte pouze jeden časovač. Interval červené a zelené určíme např. třemi událostmi **OnTimer** a interval oranžové jednou událostí **OnTimer**. Řízení změny barev budeme provádět celočíselnou proměnnou inkrementovanou událostí **OnTimer**. Při hodnotě 3 zhasneme červenou a rozsvítíme oranžovou a zelenou, při hodnotě 4 zhasneme oranžovou atd. a při hodnotě 7 proměnnou vynulujeme.
12. Přidejte ještě další semafor pro druhý směr.
13. Dále se seznámíme s možnostmi výběru komponent na formuláři. Jednotlivé komponenty vybíráme kliknutím myši. Více komponent vybíráme myší při stisknuté klávese Shift. Více komponent, které leží na ploše pomyslného obdélníka vybereme myší při stisknuté klávese Ctrl (stiskneme tlačítko myši v jednom rohu výběrového obdélníka a při stisknutém tlačítku myši přemístíme ukazatel myši do protějšího rohu, kde tlačítko

myši uvolníme). Vybrané objekty můžeme myší přetáhnout do nové pozice, vložit je do schránky (ze schránky je můžeme umístit na jiný formulář nebo panel), zrušit je apod. Vyzkoušejte.

14. Jestliže na formuláři máme několik komponent, které chceme uspořádat, můžeme využít zobrazený rastr na formuláři. Komponenty můžeme umisťovat pouze tak, aby jejich rohy byly v bodech rastru. Další možný způsob uspořádávání komponent spočívá ve výběru komponenty, podle které budeme zarovnávat a všech dalších komponent, kterých se zarovnávání má týkat. Po tomto výběru zvolíme v nabídce **Edit | Align** a v zobrazeném dialogovém okně zadáme způsob uspořádání (zadané uspořádání proběhne vzhledem k první vybrané komponentě). Obdobně můžeme způsob uspořádání zadávat z palety nástrojů zarovnávání zobrazené po volbě **View | Alignment Pallete**. Vyzkoušejte.
15. Na procvičení práce s tlačítky vytvořte tuto aplikaci. Na formulář vložte šest tlačítek. Při stisku prvního, resp. druhého překopírujte jeho vlastnost *Font* tlačítku třetímu, resp. pátému. U prvního a druhého tlačítka zadejte při návrhu jiný typ písma. Třetí tlačítko zakáže, resp. povolí první tlačítko, tj. změní jeho vlastnost *Enabled* (třetí tlačítko slouží jako přepínač; při lichém stisku zakáže a při sudém stisku tlačítko povolí; měňte i jeho titulek). Obdobně čtvrté tlačítko skryje, resp. zobrazí druhé tlačítko, tj. mění jeho vlastnost *Visible*. Páté tlačítko bude zvětšovat rozměry šestého tlačítka (při každém stisku oba jeho rozměry zvětší o jeden bod). Šesté tlačítko bude zmenšovat samo sebe (při každém stisku se oba jeho rozměry zmenší o jeden bod; testujte, aby jste se nedostali do záporných hodnot). Na všechna tlačítka vložte i vhodné titulky.

4. Další procvičování jazyka C++

1. Možnosti používání výčtových typů odpovídají jejich používání v BC++. Jejich použití si ukážeme na následujícím příkladu. Vytvoříme formulář obsahující čtyři voliče (ve skupině), tlačítko a komponentu **Label**. Nejprve na formulář (na levou stranu) vložíme **GroupBox** (jeho **Caption** změníme na „Zajímá mě“), na **GroupBox** vložíme pod sebe 4x komponentu **RadioButton** a změníme jejich vlastnosti **Caption** na *Historie*, *Literatura*, *Biologie* a *Psychologie*. Dále na formulář vložíme tlačítko s textem *Co mě zajímá* a komponentu **Label** (zde zobrazený text vyprázdníme). Nastavení voličů je vhodné určovat pomocí výčtového typu. Nadefinujeme tedy výčtový typ **TZajem** a proměnnou *Zajem* tohoto typu. Dále vytvoříme obsluhy kliknutí na jednotlivých voličích a obsluhu stisknutí tlačítka. Pokud jsme neměnili názvy komponent potom dostaneme:

```
#include <vcl\vcl.h>
#pragma hdrstop
#include "Unit1.h"
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
enum TZajem {Nic, Historie, Literatura, Biologie, Psychologie};
TZajem Zajem = Nic;
__fastcall TForm1::TForm1(TComponent* Owner)
   : TForm(Owner)
{
}
//---------------------------------------------------------------------------
void __fastcall TForm1::RadioButton1Click(TObject *Sender)
{
  Zajem = Historie;
}
//---------------------------------------------------------------------------
void __fastcall TForm1::RadioButton2Click(TObject *Sender)
{
  Zajem = Literatura;
}
//---------------------------------------------------------------------------
void __fastcall TForm1::RadioButton3Click(TObject *Sender)
{
  Zajem = Biologie;
}
//---------------------------------------------------------------------------
void __fastcall TForm1::RadioButton4Click(TObject *Sender)
{
  Zajem = Psychologie;
}
```

```
//---------------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  switch (Zajem) {
    case Historie: Label1->Caption = "Zajímám se o historii"; break;
    case Literatura: Label1->Caption = "Zajímám se o literaturu"; break;
    case Biologie: Label1->Caption = "Zajímám se o biologii"; break;
    case Psychologie: Label1->Caption ="Zajímám se o psychologii"; break;
    default: Label1->Caption = "Nezajímám se o nic";
  }
}
```

Vyzkoušejte tuto aplikaci a zjistěte jak pracuje.

2. V bodech 2.6 až 2.8 jsme vytvářeli jednoduchý textový editor. Doplňte tuto aplikaci tak, že na formulář vložíme tři voliče : *Vlevo*, *Doprostřed* a *Vpravo*, kterými budete ovlivňovat zarovnávání zobrazeného textu (vlastnost **Alignment**). Na formulář přidejte další tlačítko, kterým (prostřednictvím komponenty **ColorDialog**) budete měnit barvu textu v komponentě **Memo** (vlastnost **Font.Color**).
3. Dále se pokusíme více seznámit s třídou **AnsiString**. Tato třída má 6 konstruktorů (liší se typem parametrů), třída dále může používat běžné relační operátory, existuje zde také operátor indexace (umožňuje přístup k jednotlivým znakům řetězce), je zde také možno např. používat metody **Insert**, **Delete**¸ **LowerCase** a **UpperCase**. Seznamte se s těmito možnostmi používání třídy **AnsiString**. Vytvoříme aplikaci s formulářem, na který vložíme editační ovladač, dvě komponenty **Label** (jednu nad editační ovladač a druhý pod) a tlačítko. Editační ovladač vyprázdníme, u horní komponenty **Label** změníme **Caption** na *Zadej text:*, u spodní komponenty **Label** vlastnost **Caption** vyprázdníme a u tlačítka změníme text na *Změň na velká písmena*. Pro tlačítko vytvořte potřebnou obsluhu události **OnClick**.
4. Vytvořte aplikaci, ve které budete zjišťovat existenci zadaného podřetězce v zadaném řetězci (můžete použít metodu **Pos** třídy **AnsiString**). Řetězec a podřetězec zadávejte v editačních ovladačích. Vypisujte, zda podřetězec byl nebo nebyl nalezen.
5. Vytvořte aplikaci, která zadaný text v editačním ovladači upraví takto: Mezi jednotlivé znaky původního řetězce budou vloženy znaky mezer. Výsledek zobrazte v komponentě **Label**.
6. Vraťme se k našemu prvnímu programu, který jsme v vytvořili (kapitola 1 cvičení 10). Zdrojový text tohoto programu je (hlavičkový a CPP soubor):

```
//---------------------------------------------------------------------
#ifndef Unit1H
#define Unit1H
//---------------------------------------------------------------------
#include <vcl\Classes.hpp>
#include <vcl\Controls.hpp>
#include <vcl\StdCtrls.hpp>
#include <vcl\Forms.hpp>
//---------------------------------------------------------------------
class TForm1 : public TForm
{
__published:                        // IDE-managed Components
   TButton *Button1;
   void __fastcall Button1Click(TObject *Sender);
private:                            // User declarations
public:                             // User declarations
   __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------------------
extern TForm1 *Form1;
//---------------------------------------------------------------------
#endif

//---------------------------------------------------------------------
#include <vcl\vcl.h>
#pragma hdrstop
#include "Unit1.h"
//---------------------------------------------------------------------
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------
```

```
__fastcall TForm1::TForm1(TComponent* Owner)
   : TForm(Owner)
{
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  Form1->Color = clGreen;
  Caption = "Zelené okno";
}
//---------------------------------------------------------------------------
```

V uvedené jednotce je použita třída a objekt. Třída je datový typ, který spojuje data a kód do jednoho celku. Podobně jako záznam, má třída položky (každá má svůj vlastní datový typ). Třída také obsahuje kód (funkce, které nazýváme metody). Třídy také mohou obsahovat vlastnosti (tím se budeme zabývat později). V naší jednotce deklarujeme novou třídu **TForm1**, která je odvozena od třídy **TForm**. **TForm1** je datový typ a pro práci v aplikaci potřebujeme proměnnou tohoto typu. Deklarujeme tedy proměnnou **Form1** typu ukazatel na **TForm1**. **Form1** říkáme instance typu **TForm1** nebo objekt typu **TForm1**. Objekt je instance nějaké třídy. Objekt **Form1** reprezentuje samotný formulář (přesněji řečeno jedná se o ukazatel na formulář). Můžeme deklarovat více než jeden objekt nějaké třídy. Např. můžeme mít více podřízených oken v aplikací vícedokumentového rozhraní. Každý objekt obsahuje svá vlastní data a všechny objekty stejného typu používají stejný kód.

Naše aplikace obsahuje tlačítko. Třída **TForm1** má položku **Button1**, tj. přidané tlačítko. **TButton** je třída a tedy **Button1** je objekt. Pokaždé, když vložíme novou komponentu na formulář, je vložena do deklarace typu formuláře nová položka se jménem komponenty. Všechny obsluhy událostí jsou metody třídy formuláře. Když vytvoříme novou obsluhu události, pak její metoda je také deklarována v typu formuláře. **TForm1** obsahuje tedy metodu **Button1Click**. Aktuální kód této metody je uveden v souboru CPP. Prohlédněte si jednotku některé jiné aplikace a zjistěte, které položky a metody jsou použity.

7. Vytváření třídy začínáme odvozením třídy od existující třídy. Když přidáme do projektu nový formulář, pak C++ Builder jej automaticky odvozuje od **TForm** (je to definováno prvním řádkem deklarace typu třídy, v našem případě `class TForm1 : public TForm`). V okamžiku přidání formuláře do projektu je nová třída identická s typem **TForm**. Po přidání komponenty na formulář nebo po zápisu obsluhy události již identická není. Nový formulář je stále plně funkční formulář (můžeme měnit jeho velikost a umístění a můžeme jej také uzavřít). Nová třída formuláře totiž *zdědila* všechny datové položky, vlastnosti, metody a události od typu **TForm**. Třída od které odvozujeme svoji třídu (od které dědíme data a kód) se nazývá *předek* odvozené třídy. Odvozená třída je *potomek* svého předka. Prapředek všech tříd je třída **TObject**.
8. Rozsah platnosti určuje použitelnost a přístupnost datových položek, vlastností a metod třídy; všechny jsou v rozsahu platnosti třídy a jsou použitelné třídou a jejími potomky. Když zapisujeme kód do obsluhy události třídy, který se odkazuje na vlastnost, metodu nebo položku třídy samotné, pak nemusíme uvádět v odkazu jméno objektu. Např. příkaz v obsluze události pro **Form1** `Form1->Color = clGreen;` lze napsat jako `Color = clGreen;`. Rozsah platnosti třídy je rozšířen na všechny potomky třídy. Můžeme také použít jméno metody ze třídy předka k deklaraci metody ve třídě potomka. Jedná se o *předefinování* metody (metoda potom ve třídě potomka bude provádět něco jiného). Deklarace třídy obsahuje také klíčová slova **private:** a **public:** označující místa pro datové položky a metody, které chceme do kódu zapisovat přímo. Veřejnou část deklarace (část za klíčovým slovem **public**) používáme k deklarování datových položek a metod, ke kterým chceme přistupovat z jiných jednotek. K deklaracím v soukromé části (**private**) je omezen přístup pouze na tuto třídu.
9. Nyní se pokusíme vytvořit vlastní třídu (jinou než třídu formuláře) a to třídu umožňující pracovat s datumem. Předpokládejme následující deklaraci:

```
class TDatum : public TObject {
  int Den, Mesic, Rok;
public:
  TDatum(){};
  void NastavHodnotu(int D, int M, int R);
  bool Prestupny();
};
```

Naše třída se skládá ze tří položek: *Den*, *Mesic* a *Rok*, bezparametrického konstruktoru (naši třídu odvozujeme od třídy, ve které je definován bezparametrický konstruktor a v odvozené třídě jej musíme tedy definovat také) a dvou metod: *NastavHodnotu* a *Prestupny*. Funkce *NastavHodnotu* může vypadat např. takto:

```
void TDatum::NastavHodnotu(int D, int M, int R){
```

```
  Den = D;
  Mesic = M;
  Rok = R;
}
```

Nyní již můžeme deklarovat instanci třídy **TDatum** a s touto instancí pracovat.

```
TDatum  *Datum;
```

Touto deklarací jsme nevytvořili objekt, ale pouze místo pro uložení odkazu na objekt (ukazatel). Instance objektu vytváříme operátorem **new**. Následuje příklad práce s naší instancí:

```
Datum = new TDatum;
Datum->NastavHodnotu(27, 5, 1942);
…
```

Naší třídu se pokusíme použít v nějaké aplikaci. Vytvoříme formulář se dvěmi tlačítky (přiřadíme jim texty 1996 a 1997), kterými budeme určovat rok a budeme zjišťovat, zda se jedná o přestupný rok. Vytvoření objektu **Datum** budeme provádět v obsluze události **OnCreate** formuláře (vytváření formuláře - tím zajistíme, že objekt je vytvořen před jeho použitím). Na závěr (v obsluze události **OnDestroy** formuláře) objekt opět zrušíme. Následuje výpis obou programových souborů našeho formuláře:

```
//---------------------------------------------------------------------
#ifndef Unit1H
#define Unit1H
//---------------------------------------------------------------------
#include <vcl\Classes.hpp>
#include <vcl\Controls.hpp>
#include <vcl\StdCtrls.hpp>
#include <vcl\Forms.hpp>
//---------------------------------------------------------------------
class TDatum : public TObject {
  int Den, Mesic, Rok;
public:
  TDatum(){};
  void NastavHodnotu(int D, int M, int R);
  bool Prestupny();
};
//---------------------------------------------------------------------
class TForm1 : public TForm
{
__published:                        // IDE-managed Components
   TButton *Button1;
   TButton *Button2;
   void __fastcall FormCreate(TObject *Sender);
   void __fastcall Button1Click(TObject *Sender);
   void __fastcall Button2Click(TObject *Sender);
private:                            // User declarations
public:                             // User declarations
   __fastcall TForm1(TComponent* Owner);
};
TDatum *Datum;
//---------------------------------------------------------------------
extern TForm1 *Form1;
//---------------------------------------------------------------------
#endif


//---------------------------------------------------------------------
#include <vcl\vcl.h>
#pragma hdrstop

#include "Unit1.h"
//---------------------------------------------------------------------
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
   : TForm(Owner)
```

```
{
}
//---------------------------------------------------------------------------
void TDatum::NastavHodnotu(int D, int M, int R){
  Den = D;
  Mesic = M;
  Rok = R;
}
bool TDatum::Prestupny(){
  if (Rok % 4 != 0) return false;
  else
    if (Rok % 100 != 0) return true;
    else
      if (Rok % 400 != 0) return false;
      else return true;
}
//---------------------------------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  Datum = new TDatum;
}
//---------------------------------------------------------------------------
void __fastcall TForm1::FormDestroy(TObject *Sender)
{
  delete Datum;
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  Datum->NastavHodnotu(1, 1, 1996);
  if (Datum->Prestupny()) Caption = "Přestupný";
  else Caption = "Nepřestupný";
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{
  Datum->NastavHodnotu(1, 1, 1997);
  if (Datum->Prestupny()) Caption = "Přestupný";
  else Caption = "Nepřestupný";
}
//---------------------------------------------------------------------------
```

Prostudujte si tento výpis a zjistěte, co naše aplikace provádí. Vyzkoušejte.

10. Pokuste se nyní vynechat příkaz v obsluze události **OnCreate** formuláře (objekt Datum nebudeme vytvářet). Aplikaci znovu přeložíme a teprve při stisku některého tlačítka je signalizována chyba, která indikuje přístup k neplatnému ukazateli. Vyzkoušejte.
11. V naší třídě používáme bezparametrický konstruktor. Bylo by ale výhodné, aby náš konstruktor zároveň provedl inicializaci datových položek třídy a to podobně jako metoda *NastavHodnotu*. Vytvoříme tento konstruktor a změníme také obsluhu události **OnCreate** formuláře. Bude nyní tvořena příkazem:
    ```
    Datum = new TDatum(1, 1, 1900);
    ```
    Vyzkoušejte.
12. Definici naší třídy také můžeme umístit do samostatné jednotky a přidáme nové metody. Začneme s vývojem nové aplikace, formulář zatím necháme prázdný a zvolíme **File | New Unit** a dostaneme tento kód:
    ```
    //---------------------------------------------------------------------------
    #include <vcl\vcl.h>
    #pragma hdrstop
    #include "Unit2.h"
    //---------------------------------------------------------------------------
    ```
    Jednotku přejmenujeme na **Datumy** a zapíšeme do ní konečnou verzi definice třídy **TDatum**. Dostaneme tedy tyto dva soubory (H a CPP):
    ```
    //---------------------------------------------------------------------------
    #ifndef DatumyH
    ```

```
#define DatumyH
//---------------------------------------------------------------------------
class TDatum : public TObject {
public:
  TDatum(int D, int M, int R);
  void NastavHodnotu(int D, int M, int R);
  bool Prestupny();
  void Zvetsi();
  void Zmensi();
  void Pricti(int PocetDni);
  void Odecti(int PocetDni);
  AnsiString ZiskejText();
protected:
  int Den, Mesic, Rok;
  int DniVMesici();
};
#endif


//---------------------------------------------------------------------------
#include <vcl\vcl.h>
#pragma hdrstop

#include "Datumy.h"
//---------------------------------------------------------------------------
TDatum::TDatum(int D, int M, int R){
  Den = D;
  Mesic = M;
  Rok = R;
}
void TDatum::NastavHodnotu(int D, int M, int R){
  Den = D;
  Mesic = M;
  Rok = R;
}
bool TDatum::Prestupny(){
  if (Rok % 4 != 0) return false;
  else
    if (Rok % 100 != 0) return true;
    else
      if (Rok % 400 != 0) return false;
      else return true;
}
int TDatum::DniVMesici(){
  switch (Mesic) {
    case 1:  case 3:  case 5:  case 7:  case 8:  case 10:
    case 12: return 31;
    case 4:  case 6:  case 9:
    case 11: return 30;
    case 2: if (Prestupny()) return 29;
            else return 28;
  };
}
void TDatum::Zvetsi(){
  if (Den < DniVMesici()) Den++;
  else if (Mesic < 12) { Mesic++; Den = 1; }
       else { Rok++; Mesic = 1; Den = 1; }
}
void TDatum::Zmensi(){
  if (Den > 1) Den--;
  else if (Mesic > 1) { Mesic--; Den = DniVMesici(); }
       else { Rok--; Mesic = 12; Den = DniVMesici(); }
}
void TDatum::Pricti(int PocetDni) {
  for (int N = 1; N <= PocetDni; N++) Zvetsi();
```

```
}
void TDatum::Odecti(int PocetDni) {
  for (int N = 1; N <= PocetDni; N++) Zmensi();
}
AnsiString TDatum::ZiskejText() {
  char pom[30];
  sprintf(pom, "%d.%d.%d", Den, Mesic, Rok);
  return AnsiString(pom);
}
```

Abychom tuto jednotku mohli vyzkoušet vložíme na již vytvořený formulář komponentu **Label** (zvětšíme velikost písma) a čtyři tlačítka (vybavíme je texty **Další**, **Předchozí**, **Za 10** a **Před 10**). Objekt třídy **TDatum** vložíme jako soukromou položku do třídy formuláře. Vytvoříme obsluhy několika událostí a dostaneme:

```
//---------------------------------------------------------------------------
#ifndef Unit1H
#define Unit1H
//---------------------------------------------------------------------------
#include <vcl\Classes.hpp>
#include <vcl\Controls.hpp>
#include <vcl\StdCtrls.hpp>
#include <vcl\Forms.hpp>
//---------------------------------------------------------------------------
class TForm1 : public TForm
{
__published:                // IDE-managed Components
   TLabel *Label1;
   TButton *Button1;
   TButton *Button2;
   TButton *Button3;
   TButton *Button4;
   void __fastcall FormCreate(TObject *Sender);
   void __fastcall Button1Click(TObject *Sender);
   void __fastcall Button2Click(TObject *Sender);
   void __fastcall Button3Click(TObject *Sender);
   void __fastcall Button4Click(TObject *Sender);
private:                    // User declarations
    TDatum *Datum;
public:                     // User declarations
   __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern TForm1 *Form1;
//---------------------------------------------------------------------------
#endif


//---------------------------------------------------------------------------
#include <vcl\vcl.h>
#pragma hdrstop

#include "Datumy.h"
#include "Unit1.h"
//---------------------------------------------------------------------------
----
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
----
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
}
//---------------------------------------------------------------------------
----
void __fastcall TForm1::FormCreate(TObject *Sender)
```

```
{
  Datum = new TDatum(14, 2, 1995);
  Label1->Caption = Datum->ZiskejText();
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  Datum->Zvetsi();
  Label1->Caption = Datum->ZiskejText();
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{
  Datum->Zmensi();
  Label1->Caption = Datum->ZiskejText();
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Button3Click(TObject *Sender)
{
  Datum->Pricti(10);
  Label1->Caption = Datum->ZiskejText();
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Button4Click(TObject *Sender)
{
  Datum->Odecti(10);
  Label1->Caption = Datum->ZiskejText();
}
//---------------------------------------------------------------------------
void __fastcall TForm1::FormDestroy(TObject *Sender)
{
  delete Datum;
}
//---------------------------------------------------------------------------
```

Prostudujte si text této aplikace a pokuste se pochopit jak pracuje. Aplikaci vyzkoušejte.

13. Jako jednoduchý příklad dědičnosti můžeme pozměnit předchozí aplikaci odvozením nové třídy a modifikováním jedné z jeho funkcí. Měsíc v datumu budeme vypisovat slovně, a změníme tedy metodu *ZiskejText*. Vytvoříme další třídu (zapíšeme ji do jednotky *Datumy*):

```
class TNoveDatum : public TDatum {
public:
  TNoveDatum(int D, int M, int R): TDatum(D, M, R){};
  AnsiString ZiskejText();
};
```

Nová funkce *ZiskejText* používá k výpisu data konstantní pole s názvy měsíců:

```
char JmenaMesicu[12][10] =
  {"leden", "únor", "březen", "duben", "květen", "červen",
   "červenec", "srpen", "září", "říjen", "listopad", "prosinec"};
AnsiString TNoveDatum::ZiskejText() {
  char pom[30];
  sprintf(pom, "%d. %s %d", Den, JmenaMesicu[Mesic-1], Rok);
  return AnsiString(pom);
}
```

V naší aplikaci musíme ještě změnit **TDatum** na **TNoveDatum** (v deklaraci instance a volání konstruktoru) a aplikaci můžeme vyzkoušet.

14. Můžeme přiřadit hodnotu jednoho objektu jinému objektu, jestliže oba objekty jsou stejného typu nebo jestliže typ objektu do kterého přiřazujeme je předkem typu třídy jejíž hodnotu přiřazujeme. **TObject** je prapředek všech tříd v C++ Builderu. Mnoho obsluh událostí používá parametr *Sender*, který je typu **TObject** a proto do tohoto parametru je možno přiřadit libovolný objekt. Parametr *Sender* je vždy komponenta, která je příčinou vzniku události. Toto můžeme použít v obsluze události k zjištění typu komponenty, která událost způsobila. Ukážeme si to na následující nové aplikaci. Na formulář vložíme editační komponentu a 4x komponentu **Label** (v každé nastavíme jiný typ písma, případně jinou velikost a barvu). V této aplikaci se budeme snažit přetáhnout vlastnost Font z některé komponenty **Label** do editační komponenty. Aplikace by

měla pracovat tak, že myší ukážeme na zvolenou komponentu **Label**, stiskneme tlačítko myši a při stisknutém tlačítku přemístíme ukazatel myši na editační komponentu, kde tlačítko myši uvolníme. K zajištění této činnosti musíme provést několik věcí. U komponent z nichž zahajujeme tažení, tj. u komponent **Label**, je zapotřebí nastavit vlastnost **DragMode** na hodnotu *dmAutomatic*. Tím dosáhneme toho, že z těchto komponent můžeme zahájit tažení. Dále je zapotřebí určit, kde tažení může skončit. To určujeme parametrem *Accept* (je typu **bool**) u obsluhy událostí **OnDragOver** jednotlivých komponent. U editační komponenty tedy vytvoříme obsluhu této událostí a zjistíme v ní, zda tažení začalo v některé z komponent **Label**. Pokud ano, pak zde tažení skončit může (*Accept* nastavíme na **True**), jinak tažení nebude akceptováno. U ostatních komponent obsluhu této události měnit nemusíme, neboť implicitní hodnota parametru *Accept* je **False**, tedy tažení na ostatních komponentách skončit nemůže. Obsluha událostí **OnDragOver** u editační komponenty tedy bude tvořena příkazem:

```
if (dynamic_cast<TLabel *> (Source)) Accept = True;
```

V tomto příkazu testujeme zda hodnota parametru *Source* (počátek tažení) je typu *TLabel*, tj. zda tažení začalo v některé komponentě **Label**. Dále pro editační komponentu musíme vytvořit obsluhu události **OnDragDrop** (určíme, co se má provést po ukončení tažení). Tato obsluha musí změnit typ písma v editačním ovladači na písmo komponenty **Label**, ze které jsme začali tažení. Obsluha bude tedy tvořena příkazem:

```
Edit1->Font = dynamic_cast<TLabel *> (Source)->Font;
```

Při zápisu tohoto příkazu nevíme, která komponenta **Label** byla tažena. Odkážeme se na ní pomocí `dynamic_cast<TLabel *> (Source)` a její typ písma přiřadíme `Edit1->Font`. Tím je aplikace dokončena. Můžeme ji vyzkoušet.

15. Upravte předchozí aplikaci takto: Namísto typu písma přetahujte vlastnost **Color** a tažení provádějte na komponentu **Memo**. Na formuláři vytvořte několik komponent **Label** s různými barvami a tyto barvy přetahujte (měňte barvu textu komponenty **Memo**).
16. Předchozí zadání upravte takto: Na formulář umístěte dvě komponenty **Memo** a měňte barvu té komponenty **Memo**, na kterou barvu z některé komponenty **Label** přetáhnete.
17. Vytvořte formulář s editačním ovladačem (vlastnost **Text** nastavíme na 0) a s deseti tlačítky, jejichž vlastnosti **Caption** jsou jednotlivé číslice. Při stisku některého tlačítka, přidejte příslušnou číslici na konec vlastnosti **Text** editační komponenty. Pro všechna tlačítka používejte stejnou obsluhu události stisku tlačítka. Tím jsme vytvořili základ jednoduché kalkulačky.
18. Nyní se pokusíme zlepšit vzhled naší kalkulačky. Editační ovladač nahradíme komponentou **Panel** na kterou vložíme **Label**. U komponenty **Panel** vyprázdníme vlastnost **Caption** a změníme její barvu. U komponenty **Label** změníme vlastnost **Alignment** na **taRightJustify**, vlastnost **AutoSize** změníme na **False**, a **Caption** změníme na *0*. Dále u komponenty **Label** zvětšíme písmo a změníme jeho barvu. Potlačíme také zobrazování počátečních nul (jestliže při stisku tlačítka číslice má *Label1->Caption* hodnotu *0*, pak do této vlastnosti vložíme prázdný řetězec).
19. Na formulář vytvořený v předchozím zadání přidejte další tlačítko s jehož pomocí budete nulovat komponentu **Label**, tj. provedeme příkaz `Label1->Caption = "0";`. Dále přidejte tlačítko pro zadávání desetinné tečky. Zajistěte také, aby bylo možno zadat nejvýše jednu desetinnou tečku (deklarujeme globální proměnnou typu **bool** informující o tom, zda v čísle je tečka již zobrazena).
20. Ve vytváření kalkulačky pokračujte přidáním tlačítka pro změnu znaménka (zajistěte, aby pro nulu změna znaménka nešla použít).
21. Na formulář dále přidáme tlačítka +, -, *, / a =. Pokuste se dosáhnout toho, aby naše kalkulačka pracovala jako běžná kalkulačka. Signalizací chyb se zatím nezabývejte, pouze ošetřete dělení nulou (v komponentě **Label** např. zobrazte text *Error*).
22. Přidejte dále tlačítka pro výpočet některých funkci. Můžete také přidat tlačítka pro práci s pamětí a případně některé další činnosti kalkulačky.
23. Metody obsahují speciální skrytý parametr **this**. Je možno o něm říci, že to je ukazatel na současný objekt a je používám k odkazům na položky daného objektu uvnitř metod. Deklarujeme-li více objektů stejné třídy, potom při použití některé metody této třídy na některý z těchto objektů, musí metoda pracovat pouze s daty tohoto objektu a ostatní objekty nesmí ovlivňovat. Např. při vytváření třídy **TDatum** jsme se setkali s metodou:

```
void TDatum::NastavHodnotu(int D, int M, int R){
  Den = D;
...
```

V této metodě se identifikátorem *Den* odkazujeme na položku *Den* objektu, pro který je metoda volána. Pomocí **this** to můžeme vyjádři zápisem:

```
this->Den := D;
```

Vyzkoušejte provést tuto změnu.

24. Prapředkem všech tříd je třída TObject. Podívejme se na její definici.

```
class __declspec(delphiclass) TObject {
public:
  __fastcall TObject() {}
  __fastcall Free();
  TClass __fastcall ClassType();
  void __fastcall CleanupInstance();
  void * __fastcall FieldAddress(const ShortString &Name);
  static TObject * __fastcall InitInstance(TClass cls, void *instance);
  static ShortString __fastcall ClassName(TClass cls);
  static bool __fastcall ClassNameIs(TClass cls,const AnsiString string);
  static TClass __fastcall ClassParent(TClass cls);
  static void * __fastcall ClassInfo(TClass cls);
  static long __fastcall InstanceSize(TClass cls);
  static bool __fastcall InheritsFrom(TClass cls, TClass aClass);
  static void* __fastcall MethodAddress(TClass cls,const ShortString &Name);
  static ShortString __fastcall MethodName(TClass cls, void *Address);
  ShortString __fastcall ClassName() { return ClassName(ClassType()); }
  bool __fastcall ClassNameIs(const AnsiString string)
    { return ClassNameIs(ClassType(), string); }
  TClass __fastcall ClassParent() { return ClassParent(ClassType()); }
  void * __fastcall ClassInfo() { return ClassInfo(ClassType()); }
  long __fastcall InstanceSize() { return InstanceSize(ClassType()); }
  bool __fastcall InheritsFrom(TClass aClass)
    { return InheritsFrom(ClassType(), aClass); }
  void * __fastcall MethodAddress(const ShortString &Name)
    { return MethodAddress(ClassType(), Name); }
  ShortString __fastcall MethodName(void *Address)
    { return MethodName(ClassType(), Address); }
  virtual void __fastcall AfterConstruction();
  virtual void __fastcall BeforeDestruction();
  virtual void __fastcall Dispatch(void *Message);
  virtual void __fastcall DefaultHandler(void* Message);
private:
  virtual TObject* __fastcall NewInstance(TClass cls);
public:
  virtual void __fastcall FreeInstance();
  virtual __fastcall ~TObject() {}
};
```

V tomto výpisu je deklarována třída **TObject**, která obsahuje několik metod, které můžeme aplikovat na jakýkoli objekt, včetně námi definovaných objektů. Většina metod třídy **TObject** je často používána systémem. Některé metody třídy **TObject** mohou hrát důležitou roli při psaní obecných aplikací. Např. metoda **ClassName** vrací řetězec se jménem třídy. Tuto metodu je možné aplikovat jak na objekt (instanci), tak i na třídu (datový typ), protože se jedná o statickou metodu. Někdy může být také užitečné získat odkaz na vlastní třídu nebo jejího předchůdce. To zajišťují metody **ClassType** a **ClassParent**. Jakmile známe odkaz na třídu, můžeme jej použít stejným způsobem, jako by se jednalo o objekt: např. pro volání metody **ClassName**. Další užitečná metoda je **InstanceSize**, která vrací velikost objektu. Použití některých metod této třídy si ukážeme v následující aplikaci. Na formulář umístíme okno seznamu (**ListBox**) a tlačítko. Obsluha stisknutí tlačítka bude tvořena následujícími příkazy (povšimněte si také posledního příkazu, který zabrání dalšímu stisknutí tlačítka):

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  TDatum Datum;
  Datum = new TDatum(19, 9, 1997);
  ListBox1->Items->Add("Jméno třídy: " + Datum->ClassName());
  ListBox1->Items->Add("Jméno třídy předka: " +
    Datum->ClassParent()->ClassName());
  ListBox1->Items->Add("Velikost instance: " +
    IntToStr(Datum->InstanceSize()));
  ListBox1->Items->Add(" ");
  ListBox1->Items->Add("Jméno třídy: " + ClassName());
```

```
  ListBox1->Items->Add("Jméno třídy předka: " +
    ClassParent()->ClassName());
  ListBox1->Items->Add("Velikost instance: " + IntToStr(InstanceSize()));
  Datum->Free();
  Button1->Enabled = false;
}
```

Do této aplikace musíme přidat také naší jednotku *Datumy* (ze zadání 12). Vyzkoušejte činnost aplikace a pokuste se pochopit jak pracuje (a co dělá).

5. Vytváření formulářů

1. Nejjednodušší způsob zobrazení zprávy uživateli je použití okna zpráv, které zobrazíme funkcí **ShowMessage**. Ukážeme si to na následující aplikaci: Vytvoříme nový projekt a na formulář umístíme tlačítko. Do obsluhy události **OnClick** tlačítka zapíšeme následující kód:
   ```
   ShowMessage("C++ Builder vás zdraví");
   ```
   Aplikaci spustíme a po stisku tlačítka je zobrazeno okno zpráv se zadaným textem.
2. Metoda **MessageBox** objektu aplikace zobrazuje okno zpráv obsahující zadaný text, titulek, symbol a tlačítka. Funkce má tři parametry. První parametr určuje zobrazený text, druhý parametr titulek okna a třetí parametr určuje zobrazená tlačítka v okně a ikonu okna. Možné hodnoty posledního parametru můžeme zjistit v nápovědě. Použití si ukážeme na následujícím příkladě. Vytvoříme novou aplikaci s tlačítkem. Obsluhu události stisknutí tlačítka bude tvořit příkaz:
   ```
   Application->MessageBox("Jsi tam?","Dotaz",MB_YESNOCANCEL|MB_ICONQUESTION);
   ```
   Vyzkoušejte změnit třetí parametr funkce a zjistěte, jak se tato změna projeví.
3. Ve většině případů, ale budeme chtít zjistit, kterým tlačítkem bylo okno zpráv uzavřeno (tato informace je hodnota funkce). Ukážeme si to na příkladu, kde v předchozím zadání změníme obsluhu stisku tlačítka takto:
   ```
   switch (Application->MessageBox("Chcete aplikaci ukončit?", "Dotaz",
                                   MB_YESNO | MB_ICONQUESTION)){
     case IDYES:Application->MessageBox("Ukončení aplikace","Oznámení",MB_OK);
                Close();
                break;
     case IDNO: Application->MessageBox("Aplikace pokračuje v práci",
                                        "Oznámení", MB_OK);
   }
   ```
4. V dalším příkladu si ukážeme použití šablony dialogového okna. Vytvoříme nový prázdný projekt, zvolíme **File | New**, na stránce **Dialogs** zvolíme **Password Dialog** a stiskneme **OK**. Tím jsme k našemu projektu přidali další formulář (formulář pro zadávání hesla). Zobrazte si tento přidaný formulář (šablonu) a vyberte v ní editační komponentu. Podívejte se na vlastnost **PasswordChar** této komponenty. Hodnota vlastnosti je *. Když uživatel bude zapisovat do této komponenty nějaký text, bude namísto každého zapsaného znaku zobrazen znak hvězdičky. Tyto činnosti jsou již nadefinované v použité šabloně.

   Na náš původní formulář přidáme tlačítko a vygenerujeme obsluhu stisku tohoto tlačítka. Do obsluhy vložíme příkaz:
   ```
   PasswordDlg->ShowModal();
   ```
   zvolíme **File | Include Unit Hdr**, vybereme *Unit2* a stiskneme **OK**. (tím přidáme do *Unit1* #**include** s *Unit2*). Nyní již aplikaci můžeme spustit a vyzkoušet zadávání hesla. V tomto příkladě jsme si ukázali použití šablon, tj. již dříve vytvořených formulářů, ve kterých můžeme i něco dodělat. Je zde také ukázáno jak zobrazit další formulář jako modální dialogové okno.
5. V době návrhu jsou dialogová okna normální formuláře. Při běhu aplikace mohou být *modální* nebo *nemodální*. Když formulář spustíme modálně, pak jej uživatel musí uzavřít dříve, než může pracovat s ostatními otevřenými formuláři. Většina dialogových oken je modálních. Při nemodálním spuštění může uživatel používat libovolný formulář na obrazovce. Můžeme např. vytvořit nemodální formulář k zobrazování stavových informací. Pokud požadujeme, aby nemodální okno bylo stále zobrazeno nad ostatními okny, nastavíme jeho vlastnost **FormStyle** na hodnotu **fsStayOnTop**. Formuláře mají dvě metody, kterými je zobrazujeme. K zobrazení formuláře v nemodálním stavu, voláme metodu **Show** a k zobrazení v modálním stavu metodu **ShowModal**.

   Vytvoříme nový prázdný projekt, na formulář přidáme tlačítko a vytvoříme obsluhu stisku tohoto tlačítka s kódem:
   ```
   AboutBox->Show();
   ```

K projektu přidáme dále ze zásobníku formulářů okno **About** (zvolíme **File | New** a na stránce **Forms** vybereme *About Box*) a volbou **File | Include File Hdr** přidáme k *Unit1* jednotku *Unit2*. Nyní již aplikaci můžeme spustit a vyzkoušet si chování nemodálního dialogového okna (okno *About* nelze uzavřít). Můžete také vyzkoušet změnu chování když vlastnost **FormStyle** okna *About* změníme na hodnotu **fsStayOnTop**.

6. Pokud v naší aplikaci změníme obsluhu tlačítka na
   ```
   AboutBox->ShowModal();
   ```
   pak okno bude zobrazeno v modálním stavu. Vyzkoušejte chování modálního okna. Způsob zobrazení neovlivňuje vzhled okna.
7. První formulář, který vytvoříme se stává hlavním formulářem projektu a je prvním formulářem vytvořeným po spuštění aplikace. Můžeme ale požadovat jiný hlavní formulář. Ke změně hlavního formuláře projektu zvolíme **Options | Project**, přejdeme na stránku **Forms** a určíme nový hlavní formulář. Na stejné stránce tohoto dialogového okna můžeme také určit, které formuláře budou automaticky vytvářeny po spuštění aplikace a pořadí jejich vytváření (vytvářené formuláře jsou uvedeny v části **Auto-Create Forms** a jsou vytvářeny v uvedeném pořadí).
8. Používání modálních a nemodálních formulářů si ukážeme v další aplikaci. Hlavní formulář bude obsahovat text (komponentu **Label**) „Toto je hlavní formulář aplikace“ a dvě tlačítka: „Otevři modální formulář“ a „Otevři nemodální formulář“. Po přidání dvou nových formulářů do projektu volbou **File | New Form** (nazveme je **Modalni** a **Nemodalni**; vlastnost **Name** formuláře) můžeme napsat obsluhy stisku tlačítek. Pro **Otevři modální formulář** bude obsluha tvořena příkazy:
   ```
   TModalni *Modal = new TModalni(this);
   Modal->ShowModal();
   Modal->Free();
   ```
   Pro druhé tlačítko to jsou příkazy:
   ```
   TNemodalni *Nemodal = new TNemodalni(this);
   Nemodal->Show();
   ```
   Po zavření modálního formuláře se ukončí metoda **ShowModal** a formulář je uvolněn z paměti. Metoda **Show** je ukončena okamžitě a formulář z paměti uvolnit nelze, neboť je stále zobrazen. Nemodální formuláře jsou z paměti uvolněny až při uzavření hlavního formuláře. Není to sice nejlepší řešení, ale zatím jej použijeme. Oba naše další formuláře budou obsahovat tlačítko **Zavřít** a text „Modální formulář“ resp. „Nemodální formulář“. Obsluha stisku tlačítek **Zavřít** je tvořena pouze voláním metody *Close*. Modální formulář bude mít ještě tlačítko „Otevři další modální formulář“ pro vytvoření dalšího (stejného) modálního formuláře (u nemodálních formulářů to není zapotřebí, neboť opakovaným stiskem tlačítka na hlavním formuláři lze vytvořit více stejných nemodálních formulářů). Obsluha stisku tohoto tlačítka je tvořena příkazy:
   ```
   TModalni *DalsiModal = new TModalni(this);
   DalsiModal->ShowModal();
   DalsiModal->Free();
   ```
   Hlavní formulář používá oba další formuláře a musíme tedy do *Unit1* vložit hlavičkové soubory *Unit2* a *Unit3* (**File | Include Unit Hdr**). U obou přidaných formulářů musíme odstranit kód, který Builder automaticky generuje pro definování objektu formuláře. V obou programových jednotkách těchto formulářů tedy musíme odstranit deklaraci *TModalni Modalni;* resp. *TNemodalni Nemodalni;* a ještě předtím je nutno zrušit automatické vytváření těchto formulářů. V dialogovém okně **Project Options** na stránce **Forms** přesuneme jména těchto dvou formulářů z levého do pravého sloupce. Nyní již můžeme naši aplikaci vyzkoušet a vytvořit několik modálních a nemodálních formulářů a zjistit, jak s nimi lze pracovat. Není ale vhodné těchto oken vytvářet mnoho (z důvodu spotřeby systémových zdrojů), maximálně asi 10.
9. Vzhled formuláře můžeme ovlivňovat např. vlastnostmi **BorderIcons** a **BorderStyle** formuláře. Změna těchto vlastností se projeví až za běhu. Pokud vytváříme dialogové okno, pak je rozumné nastavit vlastnost formuláře **BorderStyle** na hodnotu **bsDialog** (jsou odstraněna minimalizační a maximalizační tlačítka a okraj okna neumožňuje změnu velikosti okna). V závislosti na uvažovaném použití dialogového okna, je nutno do okna umístit tlačítka (např. tlačítko **OK** a **Cancel**). Nastavení vlastností komponent tlačítek (vlastnosti **Cancel** a **Default**) umožňuje volat kód obsluhy události stisku tlačítka, když uživatel stiskne klávesu Esc nebo Enter. Např. když formulář obsahuje tlačítko s nastavenou vlastností **Default** na **True**, pak stisk klávesy Enter spustí kód obsluhy události **OnClick** pro toto tlačítko, pokud některé jiné tlačítko není zaostřeno. Existuje-li jiné zaostřené tlačítko, pak stisk klávesy Enter způsobí provedení obsluhy události **OnClick** zaostřeného tlačítka.

   Modální okno bude automaticky uzavřeno, když uživatel stiskne některé tlačítko s nenulovou hodnotou vlastností **ModalResult**. Různé hodnoty této vlastnosti také umožňují určit, kterým tlačítkem bylo okno uzavřeno. Např. máme-li tlačítko **Cancel**, nastavíme jeho vlastnost **ModalResult** na **mrCancel** a pokud náš

formulář obsahuje dále tlačítko **OK**, nastavíme jeho **ModalResult** na **mrOK**. Obě tlačítka lze nyní použít k uzavření okna a hodnota vlastnosti **ModalResult** použitého tlačítka je návratová hodnota funkce **ShowModal**. Testováním hodnoty funkce **ShowModal** lze zjistit, kterým tlačítkem bylo okno uzavřeno.

V mnoha případech můžeme vytvářet tlačítka pomocí komponenty **BitBtn** pouhým nastavením vlastnosti **Kind** (určíme tím text zobrazený na tlačítku a hodnotu vlastnosti **ModalResult**). Pokuste se vytvořit nějaké modální dialogové okno (obsahující pouze několik různých tlačítek), které zobrazíte pomocí stisknutí nějakého tlačítka na hlavním formuláři a testujte, kterým tlačítkem modální okno uzavřete.

10. V další aplikaci se podrobněji seznámíme s modálními dialogovými okny. Hlavní formulář této aplikace bude obsahovat dvě komponenty **Label** s texty „Toto je první text“ a „Toto je druhý text“ a tlačítko s textem **Konfigurace …**. Tlačítkem zobrazíme naše dialogové okno. Dialogové okno (nastavíme jeho vlastnost **BorderStyle** na *bsDialog*) obsahuje dvě značky s texty „Zobrazení prvního textu“ a „Zobrazení druhého textu“ a dvě komponenty tlačítek **BitBtn** (s texty **OK** a **Cancel**). Titulek dialogového okna změníme na **Volba konfigurace**. Pokud bychom dialogové okno zobrazovali pouze metodou **ShowModal**, nebylo by to vůbec užitečné. Jeden z možných přístupů je tento: Nastavíme počáteční hodnoty dialogového okna, zobrazíme dialogové okno a bylo-li stisknuto **OK**, pak přeneseme nové hodnoty do hlavního formuláře. Obsluha stisku tlačítka v hlavním formuláři tedy může obsahovat příkazy:
```
Form2->CheckBox1->Checked = Label1->Visible;
Form2->CheckBox2->Checked = Label2->Visible;
if (Form2->ShowModal() == mrOk) {
  Label1->Visible = Form2->CheckBox1->Checked;
  Label2->Visible = Form2->CheckBox2->Checked;
}
```
Aplikace je hotova a můžeme ji vyzkoušet.

11. Tento postup použití dialogového okna není jediný. Jako alternativu můžeme použít nastavení hodnot dialogového okna pouze poprvé (při návrhu), uložení aktivních hodnot po každém spuštění a vrácení aktivních hodnot při opuštění okna tlačítkem **Cancel**. Mohli bychom tedy použít obsluhu:
```
bool Puv1 = Form2->CheckBox1->Checked;
bool Puv2 = Form2->CheckBox2->Checked;
if (Form2->ShowModal() == mrOk) {
  Label1->Visible = Form2->CheckBox1->Checked;
  Label2->Visible = Form2->CheckBox2->Checked;
}
else {
  Form2->CheckBox1->Checked = Puv1;
  Form2->CheckBox2->Checked = Puv2;
}
```
Vyzkoušejte. Výhodou tohoto postupu je to, že kód, který ukládá a obnovuje původní hodnoty, může být přemístěn do kódu dialogového okna. Je to vhodné v případě, kdy okno může být zobrazeno z několika míst. Jako alternativní přístup může být celý kód přemístěn do samotného dialogového okna a obsluha stisku tlačítka **OK** dialogového okna překopíruje příslušné hodnoty do hlavního formuláře.

12. V další aplikaci si ukážeme rozšiřování dialogového okna. Vyjdeme z původní verze předchozí aplikace. Nejprve musíme dialogové okno doplnit tlačítkem (**BitBtn**) *Více >>* a novými ovladači. Okno zvětšíme, nové ovladače vložíme na zvětšenou oblast formuláře a okno opět zmenšíme na původní velikost. Přidané ovladače (mimo tlačítka **Více**) jsou tedy mimo viditelnou plochu. V našem případě se např. pokusíme rozšířit formulář směrem dolů. Na zvětšenou plochu přidáme další dvě značky s texty *Kurzíva* a *Tučně* pro zadání stylu písma na hlavním formuláři. Potřebujeme ještě znát výšku rozšířeného formuláře. Obsluha stisku tlačítka **Konfigurace** na hlavním formuláři bude nyní tvořena příkazy:
```
Form2->CheckBox1->Checked = Label1->Visible;
Form2->CheckBox2->Checked = Label2->Visible;
Form2->CheckBox3->Checked = Label1->Font->Style.Contains(fsItalic);
Form2->CheckBox4->Checked = Label1->Font->Style.Contains(fsBold);
if (Form2->ShowModal() == mrOk) {
  Label1->Visible = Form2->CheckBox1->Checked;
  Label2->Visible = Form2->CheckBox2->Checked;
  TFontStyles X;
  if (Form2->CheckBox3->Checked) X << fsItalic;
  if (Form2->CheckBox4->Checked) X << fsBold;
  Label1->Font->Style = X;
  Label2->Font->Style = X;
}
```

Do deklarace typu dialogového okna vložíme soukromé položky:

```
int PuvVyska, NovaVyska;
```

Vlastnost **AutoScroll** dialogového okna musíme nastavit na **False** a pro dialogové okno vytvoříme několik obsluh událostí. Obsluha **OnCreate** bude tvořena příkazy (zapamatujeme si výšku normálního a zvětšeného okna):

```
PuvVyska = Height;
NovaVyska = sem zapíšeme výšku zvětšeného okna ;
```

Obsluhu **OnActivate** tvoří příkazy (zakážeme nové ovladače, povolíme tlačítko **Více** a nastavíme původní výšku okna):

```
Height = PuvVyska;
BitBtn3->Enabled = true;
CheckBox3->Enabled = false;
CheckBox4->Enabled = false;
```

Zbývá ještě vytvořit obsluhu tlačítka **Více**:

```
BitBtn3->Enabled = false;
CheckBox3->Enabled = true;
CheckBox4->Enabled = true;
Height = NovaVyska;
```

Aplikace je hotova a můžeme ji vyzkoušet. Zajímavější efekt dosáhneme, když novou výšku (`Height = NovaVyska;`) budeme nastavovat v cyklu:

```
for (int I = Height; I <= NovaVyska; I++) {
  Height = I;
  Update();
}
```

Vyzkoušejte.

13. V předchozí aplikaci jsme pracovali s typem *TFontStyles*. Pokud bychom pracovali s jazykem Pascal, pak bychom řekli, že se jedná o typ množina. V Builderu je definovaná šablona třídy

    ```
    Set<typ,minimum,maximum>
    ```

    která implementuje typ množina z Pascalu. Musíme specifikovat tři parametry šablony: *typ* (typ prvků množiny; obvykle char nebo výčtový typ), *minimum* (minimální hodnota, kterou množina může obsahovat; nemůže být menší než 0) a *maximum* (maximální hodnota, kterou množina může obsahovat; nemůže být větší než 255). Každá instance množiny je objekt vytvořený na základě těchto tří parametrů. Tedy následující dva typy jsou různé, neboť mají různé minimální a maximální hodnoty:

    ```
    Set <char, 'A', 'C'> s1;
    Set <char, 'X', 'Z'> s2;
    ```

    K vytvoření více instancí typu množina použijeme výraz **typedef**:

    ```
    typedef Set <char, 'A', 'Z'> VelkaPismena;
    ```

    Můžeme deklarovat a inicializovat proměnou typu množina pomocí této syntaxe:

    ```
    VelkaPismena s1;
    s1 << 'A' << 'B' << 'C';   // Inicializace
    VelkaPismena s2;
    s2 << 'X' << 'Y' << 'Z';   // Inicializace
    ```

    Tato třída má metody *Clear* (odstraňuje všechny prvky z množiny) a *Contains* (zjištění zda množina obsahuje prvek určený parametrem). Dále jsou zde překryty následující operátory: - (rozdíl dvou množin), * (průnik množin), + (sjednocení množin), != (operátor nerovnosti), == (operátor rovnosti), = (operátor přiřazení), << (přidání prvku do množiny), >> (vyjmutí prvku z množiny) a kombinované operátory -=, *= a +=. Typ použitý v předchozím zadání je deklarován takto:

    ```
    enum TFontStyle { fsBold, fsItalic, fsUnderline, fsStrikeOut };
    typedef Set<TFontStyle, fsBold, fsStrikeOut> TFontStyles;
    ```

    Vytvoříme aplikaci ukazující použití typu množina. Na formulář vložíme editační ovladač, dvě komponenty **Label** (jednu nad a druhou pod editační ovladač) a tlačítko. Editační ovladač vyprázdníme, u horní komponenty **Label** změníme **Caption** na *Zadej text:*, u spodní komponenty **Label** vlastnost **Caption** vyprázdníme a u tlačítka změníme text na *Zjisti, zda text začíná samohláskou*. Pro tlačítko vytvoříme tuto obsluhu události **OnClick**:

    ```
    Set<char, 'A', 'Z'> Samohl;
    Samohl << 'A' << 'E' << 'I' << 'O' << 'U' << 'Y';
    if (Samohl.Contains(UpCase(Edit1->Text[1])))
      Label2->Caption = "Text začíná samohláskou";
    else  Label2->Caption = "Text začíná souhláskou";
    ```

14. Následující nová aplikace vytváří seznam použitých souhlásek a seznam použitých samohlásek v zadaném textu (uvažujte pouze písmena bez diakritických znamének). Na formulář vložíme editační ovladač pro zadávání textu, tři komponenty **Label** (první nad editační ovladač a další dva pod) a tlačítko. Editační ovladač vyprázdněte, u horní komponenty **Label** změňte **Caption** na *Zadej text*:, vlastnosti **Caption** zbývajících komponent **Label** vyprázdněte a text tlačítka změňte na *Zjisti*. Pro tlačítko vytvoříme tuto obsluhu události **OnClick**:

```
typedef Set<char, 'A', 'Z'> TMnozPismen;
TMnozPismen Samohl, Souhl, PouzSouhl, PouzSamohl;
Samohl << 'A' << 'E' << 'I' << 'O' << 'U' << 'Y';
char Znak;
for (Znak = 'A'; Znak <= 'Z'; Znak++) Souhl << Znak;
Souhl -= Samohl;
int I;
for (I = 1; I <= Edit1->Text.Length(); I++){
  Znak = UpCase(Edit1->Text[I]);
  if (Souhl.Contains(Znak)) PouzSouhl << Znak;
  if (Samohl.Contains(Znak)) PouzSamohl << Znak;
}
Label2->Caption = "Seznam použitých souhlásek: ";
for (Znak = 'A'; Znak <= 'Z'; Znak++)
  if (PouzSouhl.Contains(Znak)) Label2->Caption = Label2->Caption + Znak;
Label3->Caption = "Seznam použitých samohlásek: ";
for (Znak = 'A'; Znak <= 'Z'; Znak++)
  if (PouzSamohl.Contains(Znak)) Label3->Caption = Label3->Caption+Znak;
```

Prostudujte si použití operací s množinami. Aplikaci vyzkoušejte.

15. Upravte předchozí aplikaci tak, aby se rozlišovala velikost písmen.

16. V další aplikaci si ukážeme použití nemodálního dialogového okna. Hlavní formulář obsahuje pět komponent **Label** se jmény (Pavel, Marek, Aleš, Karel a Jana). Jestliže uživatel klikne na některé jméno, změní se jeho barva na červenou; jestliže na ně uživatel klikne dvojitě, program zobrazí dialogové okno (**Form2**) se seznamem jmen, ze kterých je možno vybírat. Na formuláři je ještě tlačítko **Styl** zobrazující dialogové okno pro zadávání stylu použitého písma. Aplikace se tedy skládá ze tří formulářů: hlavního formuláře a dvou dialogových oken. Obsluha kliknutí na všech komponentách **Label** je tvořena příkazy:

```
if (dynamic_cast<TLabel *> (Sender)) {
  Label1->Font->Color = clBlack;
  Label2->Font->Color = clBlack;
  Label3->Font->Color = clBlack;
  Label4->Font->Color = clBlack;
  Label5->Font->Color = clBlack;
  dynamic_cast<TLabel *> (Sender)->Font->Color = clRed;
}
```

Obdobně obsluha **OnDoubleClick** pro všechny komponenty **Label** je tvořena příkazy (práce se seznamy bude podrobně popsána až v další kapitole; pro pochopení jsou příkazy obsluhy doplněny komentáři):

```
if (dynamic_cast<TLabel *> (Sender)) {
  // Výběr aktivního jména v seznamu
  Form2->ListBox1->ItemIndex = Form2->ListBox1->Items->
                  IndexOf(dynamic_cast<TLabel *> (Sender)->Caption);
  // Zobrazení dialogového okna a kontrola výběru
  if ((Form2->ShowModal() == mrOk) & (Form2->ListBox1->ItemIndex >= 0)){
    // Zkopíruje vybraný prvek do komponenty Label
    int X = Form2->ListBox1->ItemIndex;
    dynamic_cast<TLabel *> (Sender)->Caption =
                  Form2->ListBox1->Items->Strings[X];
  }
}
```

Dialogové okno se seznamem neobsahuje žádný kód (nebudeme pro něj vytvářet žádné obsluhy událostí). Obsahuje komponentu **ListBox** a dvě tlačítka **BitBtn** (s texty **OK** a **Cancel**; určíme je vlastností **Kind**). Do komponenty **ListBox** vložíme několik jmen (asi 10) a musí se v nich vyskytnout všechna jména uvedená na hlavním formuláři. Pro toto dialogové okno nastavíme ještě tyto vlastnosti: **ActiveControl** na *ListBox1*, **BorderStyle** na *bsDialog* a **Caption** na *Vyber jméno*. Tím je toto dialogové okno hotovo. Hlavní formulář obsahuje ještě tlačítko. Obsluha jeho stisku je tvořena příkazem:

```
Form3->Show();
```

Dialogové okno **Styl** obsahuje dvě tlačítka **BitBtn** (s texty **Použij** a **Uzavři** – budeme je definovat sami; pro nemodální okna nelze použít **Ok** a **Cancel**), tři značky s texty Kurzíva“, „Tučně“ a „Podtrženě“ a komponentu **Label** s textem „Příklad zobrazení“. Značky zde ovládají styl zobrazení komponenty **Label**. Např. pro značku *Kurzíva* vytvoříme tuto obsluhu události **OnClick**:

```
TFontStyles X = Label1->Font->Style;
if (CheckBox1->Checked) X << fsItalic;
else X >> fsItalic;
Label1.Font.Style = X;
```

Obsluhy ostatních značek jsou obdobné, vytvořte je sami (*fsBold* a *fsUnderline*). U tlačítka **BitBtn1** nastavíme tyto vlastnosti: **Caption** na *Použij*, **Default** na *True* a pro **Glyph** vybereme nebo vytvoříme vhodný obrázek. Obdobně pro **BitBtn2**: **Cancel** na *True*, **Caption** na *Uzavři* a opět vybereme obrázek pro **Glyph**. Vlastnosti formuláře nastavíme takto: **ActiveControl** na *BitBtn1*, **BorderStyle** na *bsDialog* a **Caption** na *Vyber styl*. Obsluha stisku tlačítka **BitBtn2** bude tvořit příkaz *Close* a obsluha **BitBtn1** bude obsahovat příkazy:

```
TFontStyles X = Label1->Font->Style;
Form1->Label1->Font->Style = X;
Form1->Label2->Font->Style = X;
Form1->Label3->Font->Style = X;
Form1->Label4->Font->Style = X;
Form1->Label5->Font->Style = X;
```

Zde přistupujeme k hlavnímu formuláři a musíme tedy do *Unit3* vložit příkazem **File | Include Unit Hdr** *Unit1*. Aplikaci již můžeme vyzkoušet. Povšimněte si, že stiskem tlačítka **Použij** okno neuzavřeme.

17. Komponenta **BitBtn** může také automaticky obsloužit několik bitových map. Můžeme připravit jednu bitovou mapu obsahující určitý počet obrázků (symbolů) a nastavit tento počet jako hodnotu vlastnosti **NumGlyph**. Všechny podobrázky musí mít stejnou velikost, protože celková bitová mapa je dělena na stejné části. Jestliže v bitové mapě uvedeme více symbolů, pak jsou použity podle následujících pravidel: První symbol je použit pro povolené tlačítko (implicitní zobrazení), druhý pro zakázané tlačítko, třetí je použit při kliknutí na tlačítku a čtvrtý, když tlačítko zůstane stisknuté. Chybět mohou pouze poslední nebo poslední dva symboly. Použití si ukážeme na aplikaci, kde na formulář umístíme pouze komponentu **BitBtn** a komponentu značky (**CheckBox**), kterou tlačítko budeme zakazovat a povolovat. U tlačítka nastavíme vlastnost **Glyph** na následující obrázek. Vytvoříme jej pomocí Editoru obrázků a bude mít rozměry 96 x 32 bodů.

Vlastnost **Caption** tlačítka změníme na *Pal,* vlastnost **NumGlyphs** na 3 (nastaví se automaticky), **Spacing** na *15* a tlačítko zvětšíme. Obsluha stisku tlačítka bude tvořena příkazem:

```
ShowMessage('Prásk');
```

a obsluha **OnClick** značky bude:

```
BitBtn1->Enabled = ! BitBtn1->Enabled;
```

Nyní již aplikaci můžeme vyzkoušet. Ovládejte aplikaci tak, aby jste uviděli všechny tři obrázky. Když připravujeme bitovou mapu pro tlačítko nebo jinou grafickou komponentu, pak barva, která je použitá v levém dolním rohu, je považována za transparentní (obvykle je to barva *clOlive*). Transparentní barva je průhledná barva (je vidět barva pozadí).

18. V zadání 17. druhé kapitoly byla upravena aplikace provádějící aritmetické operace se dvěma čísly. Změňte tuto aplikaci takto: Operace provádějte s reálnými čísly. Jestliže nejsou zadány oba operandy nebo když některý z operandů nemá přípustnou hodnotu, pak při stisku tlačítka operace zobrazte okno zpráv informující o chybě. K zjištění přípustnosti zadané hodnoty můžeme použít např. standardní funkci **StrToIntDef** nebo metodu **ToIntDef** (převádí řetězec na číslo a informuje o případné chybě; ukázka použití je v zadání 22 této kapitoly).
19. Tab pořadí je pořadí, ve kterém se zaostření přesouvá z komponenty na komponentu při stisku klávesy Tab. Tab pořadí je původně určeno pořadím vkládání komponent na formulář. Toto pořadí lze změnit použitím dialogového okna Edit Tab Order (zobrazíme jej volbou **Edit | Tab Order**). Někdy chceme zabránit v přechodu na některou komponentu. K odstranění některé komponenty z Tab pořadí, nastavíme vlastnost **TabStop** této komponenty na **False**. Často chceme zabránit uživateli v přístupu na některou komponentu v dialogovém okně nebo formuláři, dokud nebyla provedena nějaká akce. Zakázané komponenty jsou zobrazeny odlišně a uživatel je nemůže používat. Zakazování a povolování komponenty ovládáme její vlastností **Enabled**. Při návrhu dialogového okna, obvykle chceme specifikovat, která komponenta má mít zaostření při prvním otevření okna. K určení aktivní komponenty při návrhu používáme vlastnost **ActiveControl**

formuláře. Jestliže v této vlastnosti není definována žádná komponenta, pak počáteční zaostření získá komponenta, která je první v Tab pořadí (neuvažují se zakázané komponenty, neviditelné komponenty a komponenty s **TabStop** nastaveným na **False**). Viditelnost komponenty ovlivňujeme vlastností **Visible**. Ke změně zaostření můžeme použít metodu **SetFocus**. Např. ke změně zaostření na komponentu **Button1** použijeme příkaz:

```
Button1->SetFocus();
```

20. Jestliže již vytvořený formulář budeme chtít používat i v dalších projektech, pak je vhodné uložit formulář jako šablonu. Provedeme to volbou **Save As Template** v místní nabídce formuláře nebo použijeme **Project | Add to Repository**.
21. V následující aplikaci se zaměříme na používání událostí **OnEnter** (získání zaostření komponentou) a **OnExit** (ztracení zaostření komponentou). Na formulář umístíme tři editační ovladače, před ně umístíme komponenty **Label** s texty „&Jméno“, „&Příjmení“ a „&Heslo“, tlačítko s textem „&Kopíruj příjmení do titulku“ a komponentu **Label**, u které změníme barvu textu. U prvních tří komponent **Label** nastavíme vlastnosti **FocusControl** na příslušný editační ovladač a tím dosáhneme, že použitím urychlovací klávesy (např. Alt-J) předáme zaostření na přiřazený editační ovladač. U editačního ovladače hesla změníme vlastnost **PasswordChar** na *. Pro stisk tlačítka vytvoříme následující obsluhu události:

```
if (Edit2->Text != "") Caption = Edit2->Text;
```

Události **OnEnter** editačních událostí a tlačítka ošetříme takto:

```
void __fastcall TForm1::Edit1Enter(TObject *Sender){
  Label4->Caption = "Zadávání jména ...";
}
//---------------------------------------------------------------------
void __fastcall TForm1::Edit2Enter(TObject *Sender){
  Label4->Caption = "Zadávání příjmení ...";
}
//---------------------------------------------------------------------
void __fastcall TForm1::Edit3Enter(TObject *Sender){
  Label4->Caption = "Zadávání hesla ...";
}
//---------------------------------------------------------------------
void __fastcall TForm1::Button1Enter(TObject *Sender){
  Label4->Caption = "Tlačítko pro kopírování";
}
```

Při stisku klávesy Tab zostření vstupu cykluje mezi editačními komponentami a tlačítkem. Toto počadí je určeno hodnotami vlastností **TabOrder**. Poslední komponenta **Label** nás informuje, který ovladač má vstupní zaostření. Můžeme vyzkoušet použití urychlovacích kláves.

22. V další aplikaci se budeme zabývat vstupem čísel pomocí editační komponenty. Vytvoříme formulář s pěti editačními ovladači a před ně umístíme komponenty **Label** určující druh prováděné kontroly nebo kdy se kontrola provede (budou zde tedy texty „&Stisknutí tlačítka“, „&Ztráta zaostření”, „Změna &textu“, „Testování &vstupu“ a „&Úplný test“). Na formulář dále umístíme tlačítko s textem „Test“. U editačních komponent nastavíme ještě vlastnosti **Tag** na hodnoty 1 až 5. Vlastnost **Tag** slouží pro uložení libovolných dat, v našem případě je to číslo editační komponenty. U první editační komponenty se kontrola provádí při stisku tlačítka. Obsluha stisku tlačítka tedy bude tvořena příkazy (hodnotu -1 použijeme pro indikaci nepřípustného čísla):

```
if (Edit1->Text != ""){
  int Cislo = Edit1->Text->ToIntDef(-1);
  if (Cislo == -1){
    Edit1->SetFocus();
    ShowMessage("V prvním editačním ovladači není číslo");
  }
  else ShowMessage("První editační ovladač v pořádku");
}
```

U druhé editační komponenty provádíme kontrolu v okamžiku, kdy komponenta ztrácí zaostření. Vytvoříme tedy obsluhu události **OnExit** této komponenty. Jelikož tuto obsluhu budeme používat i pro další editační ovladače, nemůžeme se odkazovat přímo na **Edit2**. Obsluha bude vypadat tedy takto:

```
if (dynamic_cast<TEdit *> (Sender)->Text != ""){
  int Cislo = StrToIntDef(dynamic_cast<TEdit *> (Sender)->Text,-1);
  if (Cislo == -1){
    dynamic_cast<TEdit *> (Sender)->SetFocus();
    ShowMessage("Hodnota v editačním ovladači č. " +
      IntToStr(dynamic_cast<TEdit *>(Sender)->Tag) + " je nepřípustná.");
```

```
  }
}
```

Třetí komponenta testuje správnost při každé změně. Vytvoříme tedy obsluhu události **OnChange** pro tento ovladač. Obsluha je tvořena stejnými příkazy jako předchozí obsluha. Čtvrtá komponenta povoluje zapisovat pouze číslice a stisk klávesy Backspace. Zajistíme to obsluhou události **OnKeyPress**.

```
if((Key != 8)&((Key < '0')|(Key > '9'))){
  Key = 0;
  MessageBeep(0xFFFF);
}
```

U této obsluhy, znaky které nechceme propustit do editačního ovladače změníme na znak s kódem 0 (*Key* je parametr obsluhy, který obsahuje přečtený znak). Parametr u funkce **MessageBeep** způsobuje pípnutí. U posledního editačního ovladače jsou najednou použity obsluhy z druhého až čtvrtého ovladače (nezapisujte je znova, pouze je ovladači přiřaďte). Aplikaci vyzkoušejte a v dalších vytvářených aplikacích používejte vhodné kontroly při zadávání dat.

23. Vytvořte aplikaci, kde na formulář umístíte editační ovladač a zajistíte, aby ovladač přijímal pouze písmena anglické abecedy. Případná malá písmena změňte na velká.
24. V další aplikaci na formulář umístíme editační ovladač, značky „Bold“, „Italic“ a „Underlined“ a voliče „Times New Roman“, „Arial“ a „Courier“. Značkami a voliči se pokusíme ovládat písmo použité v editační komponentě. Obsluhy kliknutí na značkách vyřešte sami. Obsluha kliknutí na voliči **Courier** může být např. tvořena příkazem:

    ```
    Edit1->Font->Name = "Courier";
    ```

    Dokončete tuto aplikaci sami.
25. Když uživatel stiskne některé tlačítko myši, pokud ukazatel myši je nad formulářem (nebo nad některou komponentou) vzniká událost **OnMouseDown**. Obdobně při uvolnění tlačítka je generována událost **OnMouseUp**. Další událost se vztahuje k pohybu myši. Je to událost **OnMouseMove**. Toto jsou základní události týkající se myši. Do této skupiny událostí patří i událost **OnClick**, kterou jsme již často používali. Událost kliknutí se týká pouze levého tlačítka myši, ale ostatní z těchto událostí se týkají i dalších tlačítek. Myš může mít dvě nebo tři tlačítka (prostřední tlačítko myši se většinou nepoužívá, neboť nemáme jistotu, že všichni uživatelé naší aplikace budou vlastnit třítlačítkovou myš).

    Obsluha události **OnMouseDown** má pět parametrů. První z nich je obvyklý parametr **Sender**. Další parametr **Button** indikuje, které tlačítko myši bylo stisknuto. Možné hodnoty tohoto parametru jsou: *mbRight*, *mbLeft* nebo *mbCenter*. Třetí parametr (**Shift**) určuje, které klávesy (z Alt, Ctrl nebo Shift) byly při výskytu události stisknuty. Tento parametr je typu množina (najednou může být stisknuto více kláves). Poslední dva parametry (**X** a **Y**) určují souřadnice (v souřadném systému uživatelské plochy okna) ukazatele myši v okamžiku stisku tlačítka myši.

    Začneme novou aplikaci. Formulář této aplikace neobsahuje žádné komponenty. Vytvoříme obsluhu události **OnMouseDown** formuláře, která je tvořena příkazem:

    ```
    if (Button == mbLeft) Canvas->Ellipse(X-10, Y-10, X+10, Y+10);
    ```

    Jestliže při spuštěné aplikaci uživatel stiskne levé tlačítko myši, je na formuláři nakreslen kruh. Pro nakreslení kruhu používáme metodu *Ellipse*, neboť metoda pro nakreslení kruhu neexistuje. Metoda *Ellipse* vyžaduje čtyři parametry, reprezentující protilehlé strany opsaného obdélníku.

    Kreslení provádíme na plátno formuláře, tj. na jeho vlastnost **Canvas**. Objekt **Canvas** má dvě odlišné funkce: obsahuje sbírku nástrojů pro kreslení (je to pero a štětec) a má řadu kreslících metod, které používají aktivní nástroje.

    Pokuste se rozšířit funkci uvedené obsluhy takto: Jestliže při stisknutí tlačítka myši je stisknuta klávesa Shift, pak nekreslete kruh, ale čtverec; objekt **Canvas** má pro nakreslení obdélníku metodu *Rectangle*, kterou můžeme použít i k nakreslení čtverce. Podmínka zjišťující, zda je stisknutá klávesa Shift je

    ```
    (Shift.Contains(ssShift))
    ```

    Vytvoříme ještě obsluhu události **OnMouseMove**, tvořenou příkazy:

    ```
    char Pom[30];
    sprintf(Pom, "Souřadnice: x = %d, y = %d", X, Y);
    Caption = AnsiString(Pom);
    ```

    která nás informuje o souřadnicích ukazatele myši. Dokončete tuto aplikaci a vyzkoušejte. Povšimněte si, že když nakreslíme několik tvarů na formulář a formulář překryjeme jiným oknem, pak po odstranění překrývajícího okna se naše kresba neobnoví. Je to implicitní chování oken. Ve vývoji této aplikace budeme pokračovat, nejprve se ale seznámíme s vytvářením nabídky.
26. Vytvářenou aplikaci můžeme také vybavit nabídkou. Na formulář musíme nejdříve přidat komponentu **MainMenu** nebo **PopupMenu**. Komponenta **MainMenu** vytváří řádek nabídky a komponenta **PopupMenu**

vytváří místní nabídku (vyvoláváme ji kliknutím pravého tlačítka myši). Jestliže na komponentě nabídky dvojitě klikneme, pak otevřeme Návrhář nabídky. V návrháři nabídky zadáváme texty prvků nabídky, které chceme zobrazovat v nabídce (jedná se o vlastnosti **Caption** prvků nabídky). Vlastnosti **Name** prvků nabídky se vytvářejí sami, ale můžeme je změnit (pokud používáme písmena s diakritickými znaménky je často jméno prvku nabídky nesrozumitelné). K vložení nového prázdného prvku vybereme v návrháři nabídky prvek, před který chceme prvek vložit a stiskneme klávesu Insert. Ke zrušení prvku nabídky, prvek vybereme a stiskneme klávesu Delete. Chceme-li do nabídky vložit oddělovací řádek použijeme prvek nabídky s hodnotou vlastnosti **Caption** rovnou znaku pomlčky. Pro specifikaci urychlovací klávesy přidáme znak & před znak ve vlastnosti **Caption**, který chceme zobrazovat podtrženě. Ke specifikaci zkracovací klávesy použijeme vlastnost **ShortCut** prvku nabídky. Builder netestuje duplicitu zkracovacích nebo urychlovacích kláves.

K vytvořené nabídce potřebujeme ještě vytvořit přiřazené příkazy. Každý prvek nabídky má událost **OnClick**. Vytvořením obsluhy této události přiřadíme k prvku nabídky kód, který má být proveden při volbě tohoto prvku v nabídce (nebo při použití odpovídající zkracovací klávesy). Pro generování obsluhy události pro prvek nabídky dvojitě klikneme na prvek nabídky v okně Návrháře nabídky. Pomocí Inspektora objektů můžeme také k události **OnClick** přiřadit již existující obsluhu. Prostřednictvím vlastností **Visible** prvku nabídky můžeme prvek nabídky ukrýt a vlastností **Enabled** lze prvek nabídky zakázat. Použitím metody **Insert** lze k nabídce přidat další prvek.

K vytvoření podnabídky, vybereme prvek nabídky, u kterého chceme vytvořit vnořenou nabídku a stiskneme Ctrl+→. Během návrhu můžeme také prvky nabídky přetahovat. Během návrhu můžeme vytvořenou nabídku zobrazit (i bez spuštění aplikace). K zobrazení nabídky přejdeme na formulář jehož nabídku chceme zobrazit a je-li v tomto formuláři více nabídek vybereme požadovanou nabídku v jeho vlastnosti **Menu**. Prvky nabídky můžeme také editovat přímo v Inspektoru objektů.

Zadání 2 z kapitoly 4 upravte takto: Z formuláře odstraňte všechna tlačítka a voliče, komponentu **Memo** zvětšete na celý formulář a ovládání aplikace provádějte pomocí nabídky (funkce tlačítek a voličů převeďte na volby v nabídce).

27. V další aplikaci se budeme zabývat změnami v nabídce. Při modifikaci prvku nabídky se obvykle používají tři vlastnosti. Vlastnost **Checked** používáme k přidání nebo odstranění značky odškrtnutí v prvku nabídky. Vlastnost **Enabled** je možno použít k znevýraznění prvku nabídky a uživatel jej potom nemůže zvolit. Dále můžeme měnit vlastnost **Caption** prvku nabídky, tzn. zobrazený text prvku nabídky. Použití těchto vlastností si ukážeme v této aplikaci. Na formulář umístíme dvě komponenty **GroupBox** (nad sebe) a vedle každé z nich umístíme tlačítko (s textem **Ukryj**). Do horního **GroupBox** umístíme dva editační ovladače (vložíme do nich nějaké implicitní texty) a změníme jeho **Caption** na *Editační ovladače*. Do spodního **GroupBox** umístíme dvě značky a změníme jeho **Caption** na *Značky*. Na formulář přidáme ještě nabídku a to podle následující tabulky:

&Soubor	&Tlačítka	&Zobrazit	&Nápověda
&Konec	Zakaž &první	&Editační ovladače	&O aplikaci
	Zakaž &druhé	Znač&ky	

U prvku nabídky *Zakaž první* nastavíme vlastnost **Enabled** na **False** a u obou prvků v nabídce *Zobrazit* nastavíme **Checked** na **True**. Komponenty uvnitř **GroupBox** se v naši aplikaci nevyužívají, tlačítka budeme používat k zobrazení nebo skrytí jednotlivých **GroupBox** i s ovladači, které obsahují. Stejnou akci je možné provést příkazy v nabídce **Zobrazit**. Pokaždé, když zvolíme jeden z těchto příkazů nebo stiskneme jedno z těchto tlačítek, provedou se tři akce: změní se viditelnost příslušného **GroupBox**, změní se text na tlačítku z **Ukryj** na **Zobraz** (nebo naopak) a změní se stav odškrtnutí prvku nabídky. Obsluha stisku prvního tlačítka a volby *Editační ovladače* bude tedy tvořena příkazy (pro obě akce použijte stejnou obsluhu události):

```
GroupBox1->Visible = ! GroupBox1->Visible;
Editanovladae1->Checked = ! Editanovladae1->Checked;
if (GroupBox1->Visible) Button1->Caption = "Ukryj";
else Button1->Caption = "Zobraz";
```

Obdobně vytvořte i obsluhu události pro druhé tlačítko a volbu *Značky*. Dále se budeme zabývat volbami v nabídce *Tlačítka*. Základní myšlenkou je, aby tyto volby měnily text podle prováděné akce. Toho dosáhneme např. obsluhou:

```
if (Button1->Enabled){
  Button1->Enabled = false;
  Zakaprvn1->Caption = "Povol &první";
} else {
  Button1->Enabled = true;
  Zakaprvn1->Caption = "Zakaž &první";
```

```
}
```

Situace je ale složitější protože požadujeme aby druhé tlačítko mohlo být povoleno, pouze pokud je povoleno první tlačítko. Mohou být tedy povoleny obě tlačítka, žádné tlačítko nebo první povoleno a druhé zakázáno. Když jsou obě tlačítka povolena, zakážeme volbu nabídky **Zakaž první**, čímž zabráníme uživateli v zakázání prvního tlačítka. Když jsou zakázána obě tlačítka, zakážeme volbu nabídky **Zakaž druhé**, čímž zabráníme v povolení pouze druhého tlačítka. Předchozí obsluhu tedy nahradíme obsluhou:

```
if (Button1->Enabled){
  Button1->Enabled = false;
  Zakaprvn1->Caption = "Povol &první";
  Zakadruh1->Enabled = false;
} else {
  Button1->Enabled = true;
  Zakaprvn1->Caption = "Zakaž &první";
  Zakadruh1->Enabled = true;
}
```

Obsluha volby **Zakaž druhé** je obdobná

V této aplikaci se zabýváme změnou prvku nabídky. Mohli bychom také zneviditelnit prvek nabídky změnou hodnoty vlastnosti **Visible** na **False**. Je ale vhodné tuto operaci nepoužívat. Doplňte zbývající obsluhy a aplikaci vyzkoušejte. V této aplikaci vidíme, že vlastnosti **Name** jednotlivých prvků nabídek jsou poněkud „nečitelné“. Je tedy vhodné je změnit na snadněji zapamatovatelná jména.

28. Vlastnost **Visible** je ale možno používat ke změně jednotlivých nabídek. Můžeme si vybrat mezi zakázanými nebo neviditelnými podnabídkami. Druhý přístup je častější. Do nabídky předchozí aplikace doplníme další nabídku *Nabídky* s volbami: „Odstraň soubor“, „Odstraň nápovědu“, „Zakaž tlačítka“ a „Zakaž zobrazit“. Obsluha první volby bude tvořena příkazy:

```
Soubor1->Visible = ! Soubor1->Visible;
Odstrasoubor1->Checked = ! Odstrasoubor1->Checked;
```

Vytvořte zbývající obsluhy a aplikaci vyzkoušejte.

29. V jedné aplikaci můžeme také střídat několik nabídek. Pokusíme se to vyzkoušet. Vytvoříme aplikaci, ve které na formulář vložíme dvě komponenty **MainMenu** (pro každou z těchto komponent navrhneme jinou nabídku). Právě používaná nabídka formuláře je určena vlastností **Menu** formuláře. Můžeme tedy v první nabídce mít prvek nabídky s obsluhou:

```
Form1->Menu := MainMenu2;
```

Volbou tohoto prvku zobrazíme druhou nabídku. Obdobně v druhé nabídce budeme mít volbu pro zobrazení první nabídky. Dokončete tuto aplikaci a vyzkoušejte.

30. Před naším seznamováním s nabídkou jsme vytvořili aplikaci, kde jsme myší kreslili na formuláři čtverce a kruhy. Nyní k této aplikaci přidáme nabídku pro volbu barvy tvarů a jejich okrajů (tím se myslí barva štětce a pera) a pro výběr velikosti tvaru a jeho okraje. Naši aplikaci vybavíme touto nabídkou (urychlovací klávesy si doplňte sami; doplňujte je také v dalších aplikacích):

Soubor	Barva	Velikost	Nápověda
Nový	Pero	Zvětšit šířku pera	O aplikaci
-----	Štětec	Zmenšit šířku pera	
Konec		----------	
		Zvětšit velikost tvaru	
		Zmenšit velikost tvaru	

Formulář inicializuje a ukládá pouze současný poloměr kruhu (tato hodnota se také používá jako polovina strany čtverce), ostatní hodnoty jsou uloženy ve vlastnostech plátna (**Canvas**). Jako soukromou položku formuláře tedy přidáme:

```
int Polomer;
```

Pro změnu barev použijeme komponentu **ColorDialog**. Budeme se také snažit, aby rozměry nebyly záporné. Obsluha události **OnCreate** formuláře bude tvořena příkazem:

```
Polomer = 5;
```

Obsluhu události **OnMouseDown** změníme pouze tak, že namísto konstant 10 použijeme položku *Polomer*. Obsluha události **OnMouseMove** zůstane beze změny. Obsluha volby v nabídce **Barva | Pero** bude tvořena příkazy:

```
ColorDialog1->Color = Canvas->Pen->Color;
if (ColorDialog1->Execute()) Canvas->Pen->Color = ColorDialog1->Color;
```

Obdobně vyřešíme i změnu barvy štětce (**Brush**). Obsluha volby **Velikost | Zmenšit šířku pera** bude:

```
Canvas->Pen->Width = Canvas->Pen->Width - 2;
```

```
if (Canvas->Pen->Width < 3) Zmenitkupera1->Enabled = false;
```

a obsluha volby **Velikost | Zvětšit šířku pera** bude:

```
Canvas->Pen->Width = Canvas->Pen->Width + 2;
Zmenitkupera1->Enabled = true;
```

Obdobně vyřešte volby změny velikosti tvaru (položky *Polomer*). Tuto hodnotu měňte s krokem 5. Volbou **Nápověda | O aplikaci** zobrazte okno zpráv funkcí *ShowMessage* (zobrazte zde název aplikace a svoje jméno). Obsluha volby **Soubor | Konec** bude tvořena příkazem:

```
Close();
```

Zbývající nevyřešená volba v nabídce (**Soubor | Nový**). Je tvořena příkazem:

```
Refresh();
```

Tato metoda překreslí celé okno a smaže jeho obsah. Tím ale ztratíme svoji kresbu. Aplikaci vyzkoušejte.

31. Předchozí aplikace má stále ten nedostatek, že překrytím okna nebo zmenšením jeho velikosti ztrácíme jeho obsah. Nyní se tento problém pokusíme vyřešit. Nejprve se ale musíme seznámit s tím, co je ve Windows kreslení a malování. Kreslení (drawing) je to, co jsme zatím prováděli v naši aplikaci (aplikace neví jak vytvořený obrázek překreslit). Malování (painting) je to, co musíme dělat, aby aplikace překreslila za jakýchkoli podmínek nakreslený obrázek. Jestliže nabídneme metodu pro překreslení obsahu formuláře (obsluha události **OnPaint**), pak tato metoda je automaticky volána, když část formuláře byla skryta a potřebuje překreslit.

    Vyřešení našeho problému spočívá v ukládání informací o nakreslených tvarech a vytvoření obsluhy události **OnPaint**, ve které naše tvary zobrazíme. Pro ukládání informací o nakresleném tvaru si vytvoříme novou strukturu:

```
struct TTvar {
  bool Kruznice;
  int X, Y, Velikost, PeroSire;
  TColor BarvaPera, BarvaStetce;
};
```

    Deklaraci této struktury umístíme do hlavičkového souboru formuláře před deklaraci typu formuláře. Proměnné této struktury (popisující jednotlivé tvary) budeme ukládat do seznamu (standardní třída **TList**). Tento seznam nazvaný **SeznamTvaru** je soukromá položka formuláře a je inicializována v obsluze události **OnCreate** formuláře:

```
Polomer = 5;
SeznamTvaru = new TList;
```

    Tvar do seznamu budeme přidávat v obsluze události **OnMouseDown** a po přidání tvaru do seznamu vždy provedeme překreslení. Tato událost bude nyní obsahovat příkazy:

```
TTvar *Tvar;
if (Button == mbLeft) {
  Tvar = new TTvar;
  Tvar->Kruznice = ! Shift.Contains(ssShift);
  Tvar->X = X;
  Tvar->Y = Y;
  Tvar->Velikost = Polomer;
  Tvar->PeroSire = Canvas->Pen->Width;
  Tvar->BarvaPera = Canvas->Pen->Color;
  Tvar->BarvaStetce = Canvas->Brush->Color;
  SeznamTvaru->Add(Tvar);
  Repaint();
}
```

    Aby probíhalo překreslování, je nutno vytvořit obsluhu události **OnPaint**:

```
TTvar *Tvar;
Tvar = new TTvar;
for (int I=0; I < SeznamTvaru->Count; I++){
  Tvar = (TTvar *)SeznamTvaru->Items[I];
  Canvas->Pen->Color = Tvar->BarvaPera;
  Canvas->Pen->Width = Tvar->PeroSire;
  Canvas->Brush->Color = Tvar->BarvaStetce;
  if (Tvar->Kruznice) Canvas->Ellipse(Tvar->X-Tvar->Velikost,
            Tvar->Y-Tvar->Velikost,Tvar->X+Tvar->Velikost,
            Tvar->Y+Tvar->Velikost);
  else Canvas->Rectangle(Tvar->X-Tvar->Velikost,
            Tvar->Y-Tvar->Velikost,Tvar->X+Tvar->Velikost,
            Tvar->Y+Tvar->Velikost);
```

```
}
delete Tvar;
```

V této obsluze jsou vykresleny všechny tvary uvedené v seznamu (počet prvků seznamu je určen **Count** a přistupujeme k nim indexováním vlastnosti **Items**). Musíme ještě změnit obsluhu volby nabídky **Soubor | Nový** a to takto:

```
if (SeznamTvaru->Count > 0)
  if (Application->MessageBox("Chcete opravdu zrušit všechny tvary?",
                    "Dotaz", MB_YESNO | MB_ICONQUESTION) == IDYES){
    SeznamTvaru->Clear();
    Refresh();
  }
```

Aplikaci vyzkoušejte. Přidejte ještě volbu nabídky pro změnu barvy formuláře.

32. Komponenta **Image** je obvykle považována za zobrazovač obrázků. Vytvoříme nyní aplikaci pro zobrazování obrázků. Na formulář umístíme komponentu **Image** (bude zabírat celou plochu formuláře) a vytvoříme následující nabídku:

```
Soubor          Volby           Nápověda
Otevřít         Zaplnit         O aplikaci
-----           Centrovat
Konec
```

Obsluhu voleb **Konec** a **O aplikaci** si vytvořte sami. Volba **Otevřít** bude tvořena kódem:

```
if (OpenDialog1->Execute()){
  Image1->Picture->LoadFromFile(OpenDialog1->FileName);
  Caption = "Obrázek: " + OpenDialog1->FileName;
}
```

Na formulář musíme tedy vložit i komponentu **OpenDialog**, kde nastavíme vlastnosti **Filter** na *Bitové mapy (*.bmp)*, *Ikony (*ico)* a *MetaSoubor (*.wmf)* a **Options** na [ofHideReadOnly, ofPathMustExist, ofFileMustExist]. Zjistěte, co tyto volby znamenají. Volba nabídky **Zaplnit** bude tvořena příkazy:

```
Image1->Stretch = ! Image1->Stretch;
Zaplnit1->Checked = Image1->Stretch;
```

a volba **Centrovat** příkazy:

```
Image1->Center = ! Image1->Center;
Centrovat1->Checked = ! Image1->Center;
```

Nyní již aplikaci můžeme vyzkoušet. Zjistěte, co provádějí jednotlivé volby nabídky.

33. Poslední dvě aplikace nyní spojíme dohromady. Můžeme načíst existující bitovou mapu, nakreslit přes ní několik tvarů a výsledek uložit do souboru. V této aplikaci použijeme předposlední verzi aplikace kreslení tvarů (informace o nakreslených tvarech nemusíme ukládat, neboť jsou vloženy přímo do bitové mapy). Všechny výstupní operace týkající se plátna formuláře musíme změnit na plátno komponenty **Image** (např. *Image1->.Canvas->Pen->Color*). Aplikace bude mít nabídku:

```
Soubor          Barva       Velikost                Nápověda
Nový            Pero        Zvětšit šířku pera      O aplikaci
Otevřít         Štětec      Zmenšit šířku pera
Uložit jako                 ------------------
-----                       Zvětšit velikost tvaru
Konec                       Zmenšit velikost tvaru
```

Volba **Barva | Pero** bude tedy obsahovat příkazy:

```
ColorDialog1->Color = Image1->Canvas->Pen->Color;
if (ColorDialog1->Execute())
  Image1->Canvas->Pen->Color = ColorDialog1->Color;
```

Volba **Ulož jako** bude tvořena příkazem:

```
if (SaveDialog1->Execute())
  Image1->Picture->SaveToFile(SaveDialog1->FileName);
```

U komponenty **SaveDialog** musíme nastavit tyto vlastnosti: **DefaultExt** na *BMP*, **FileName** na *Tvary.bmp*, **Filter** na *Soubory bitové mapy (*.bmp)* a **Options** na *[ofOwerwritePromtp, ofHideReadOnly, ofPathMustExist, ofFileMustExist]*. U komponenty **OpenDialog** nastavíme vlastnosti: **DefaultExt** na *BMP*, **Filter** na *Soubory bitové mapy (*.bmp)* (v této aplikaci je možno používat pouze soubory bitových map) a **Options** na *[ofHideReadOnly, ofPathMustExist, ofFileMustExist, ofShareAware]*. Musíme samozřejmě uložit původní barvu a po otevření bitové mapy ji obnovit. To vyřešíme v obsluze volby **Otevřít** takto:

```
if (OpenDialog1->Execute()){
  TColor BPera = Image1->Canvas->Pen->Color;
```

```
  TColor BStetce = Image1->Canvas->Brush->Color;
  int SirePera = Image1->Canvas->Pen->Width;
  Image1->Picture->LoadFromFile(OpenDialog1->FileName);
  Caption = "Obrázek: " + OpenDialog1->FileName;
  Image1->Canvas->Pen->Color = BPera;
  Image1->Canvas->Brush->Color = BStetce;
  Image1->Canvas->Pen->Width = SirePera;
}
```

Za zmínku stojí ještě obsluha volby **Nový**. Zde je nutno plochu komponenty **Image** vybarvit bílou barvou. To provedeme takto:

```
if (Application->MessageBox("Chcete opravdu zrušit všechny tvary?",
                  "Dotaz", MB_YESNO | MB_ICONQUESTION) == IDYES){
  TRect Oblast=Rect(0,0,Image1->Picture->Width,Image1->Picture->Height);
  TColor Barva = Image1->Canvas->Brush->Color;
  Image1->Canvas->Brush->Color = clWhite;
  Image1->Canvas->FillRect(Oblast);
  Image1->Canvas->Brush->Color = Barva;
}
```

Ostatní obsluhy událostí nepotřebují žádné vysvětlení. Vytvořte je sami a aplikaci vyzkoušejte.

34. Někdy se můžeme dostat do situace, kdy všechny ovladače na formuláři se nevejdou na plochu formuláře. Jestliže komponenty umístíme do velkého formuláře a potom zmenšíme jeho velikost (tak, že některé komponenty se do něj již nevejdou), jsou do formuláře automaticky přidány posuvníky. Toto nyní vyzkoušíme. V nové aplikaci uděláme široký formulář, po celé jeho ploše rozmístíme různé komponenty a formulář zúžíme. Aplikaci spustíme a vyzkoušíme.

    Do této aplikace se dále pokusíme přidat malé stavové okno, které nás bude informovat o stavu vodorovného posuvníku. K aplikaci přidáme další formulář. Přidáme na něj (pod sebe) dvě komponenty **Label** s texty „Šíře formuláře:“ a Pozice posuvníku:“ a vedle nich další dvě komponenty **Label** pro výpis aktuálních hodnot. Velikost tohoto formuláře zmenšíme a nastavíme jeho vlastnosti takto: **Caption** na *Stav posuvníku*, **BorderIcons** na *[biSystemMenu]*, **BorderStyle** na *bsSingle*, **FormStyle** na *fsStayOnTop* a **Visible** na *True* (po spuštění aplikace bude formulář viditelný). Pro hlavní formulář vytvoříme ještě obsluhu události **OnResize** s příkazy:

    ```
    Form2->Label3->Caption = IntToStr(ClientWidth);
    Form2->Label4->Caption = IntToStr(HorzScrollBar->Position);
    ```

    Aplikaci spustíme a při změně velikosti formuláře se mění zobrazované hodnoty ve stavovém okně. Když ale použijeme posuvník, pak hodnoty ve stavovém okně se nezmění. Builder toto neumožňuje neboť pro formulář neexistuje událost **OnScroll** (není to nutné, neboť ve většině případů formuláře obsluhují posuvníky automaticky). Tato událost by byla zapotřebí pouze v některých speciálních případech (např. právě v naši aplikaci).

35. Velkou výhodou Builderu je to, že když velikost některé velké komponenty na formuláři se změní, pak se podle potřeby automaticky přidají nebo odstraní posuvníky. Jako příklad můžeme použít komponentu **Image**. Jestliže do komponenty nahrajeme nový obrázek a její vlastnost **AutoSize** je nastavena na *True*, pak komponenta automaticky mění velikost podle velikosti obrázku a formulář přidává nebo odstraňuje posuvníky. Vraťme se k naši aplikaci zobrazovače obrázků. Jedná se o formulář zobrazující bitovou mapu nahranou ze souboru, která je zobrazena v původní velikosti nebo deformována tak, aby se do formuláře vtěsnala. U našeho původního zobrazovače obrázků (zadání č. 32) změníme u komponenty **Image** vlastnost **AutoSize** na *True*, z nabídky odstraníme volbu **Centrovat** a změníme obsluhu volby **Zaplnit** (při zapnuté volbě odstraníme posuvníky) takto:

    ```
    Image1->Stretch = ! Image1->Stretch;
    Centrovat1->Checked = Image1->Stretch;
    if (Image1.Stretch) Image1->Align = alClient;
    else {
      Image1->Align = alNone;
      Image1->Height = Image1->Picture->Height;
      Image1->Width = Image1->Picture->Width;
    }
    ```

    Aplikace je hotova. Zmenšíme formulář a nahrajeme nějaký velký obrázek. Vidíme, že jsou v případě potřeby automaticky přidány posuvníky.

36. Pro aplikaci můžeme také vytvořit místní nabídku (nabídku aktivovanou stiskem pravého tlačítka myši – obvykle závisí na umístění kurzoru myši). Tyto nabídky se snadno používají, protože obsahují pouze několik voleb, vztahujících se k právě vybrané komponentě. Jestliže chceme používat místní nabídku, pak musíme

provést několik jednoduchých operací. Na formulář přidáme komponentu **PopupMenu**, Návrhářem nabídky do ní přidáme nějaké prvky (a vytvoříme k nim obsluhy) a tuto komponentu přiřadíme vlastnosti **PopupMenu** formuláře nebo některé komponenty. To je vše. Když nyní klikneme pravým tlačítkem myši na formuláři nebo komponentě (do jejichž vlastností **PopupMenu** jsme přiřadili místní nabídku) je místní nabídka zobrazena a můžeme ji použít. Vytvořte aplikaci, kde na formuláři bude pouze editační ovladač. K formuláři vytvořte místní nabídku, která umožní měnit barvu formuláře (použijte zde volby „Modrá", „Červená" a „Zelená") a pro editační ovladač vytvořte jinou místní nabídku (s volbami „Písmo" a „Velikost písmen" – touto druhou volbou měňte vlastnost **CharCase** editačního ovladače). Aplikaci vyzkoušejte.

37. Při zobrazování místní nabídky barvy formuláře by bylo vhodné, aby prvek nabídky určující současnou barvu formuláře byl odškrtnutý. Pokusíme se tuto změnu provést. Využijeme toho, že při zobrazování místní nabídky vzniká událost **OnPopup**. Vytvoříme tedy obsluhu této události:

```
for (int I = 0; I < PopupMenu1->Items->Count; I++)
  PopupMenu1->Items->Items[I]->Checked = false;
if (Color == clBlue) Modr1->Checked = true;
if (Color == clRed) erven1->Checked = true;
if (Color == clGreen) Zelen1->Checked = true;
```

Na počátku obsluhy je u všech prvků místní nabídky zrušeno odškrtnutí pomocí cyklu na poli **Items** místní nabídky. Cyklus probíhá od 0 do *Count-1*. Jeho výhodou je to, že funguje na všech prvcích nabídky, nezávisle na jejich počtu. Vyzkoušejte.

38. Zatím jsme používali automatické místní nabídky. Alternativou je nastavení vlastnosti **AutoPopup** na *False* a použití metody **Popup** pro zobrazení místní nabídky na obrazovku. Metoda vyžaduje dva parametry, **X** a **Y** souřadnice plánovaného umístění nabídky. Problém je v tom, že musíme zadat souřadnice bodu a ne uživatelské souřadnice, které se u uživatelské plochy obvykle používají.

Začneme novou aplikaci, kde na formulář umístíme pouze komponentu **Memo**, která zaplní celý formulář. Do komponenty **Memo** zapíšeme několik řádků textu. Pro komponentu vytvoříme dvě místní nabídky. První bude umožňovat změnu písma a barvy komponenty (prvky **Písmo** a **Barva**) a druhá bude určovat zarovnávání textu v komponentě (prvky **Vlevo**, **Vpravo** a **Doprostřed**). U obou nabídek nastavíme vlastnost **AutoPopup** na *False*. Budeme se snažit, aby vždy při stisku pravého tlačítka myši se nabídka změnila. Vytvoříme tedy obsluhu stisku tlačítka myši (**OnMouseDown**) pro komponentu **Memo**:

```
if (Button == mbRight) {
  TPoint KlBod, ObBod;
  KlBod.x  = X;
  KlBod.y = Y;
  ObBod = ClientToScreen(KlBod);
  PocetKliknuti++;
  if (PocetKliknuti & 1) PopupMenu1->Popup(ObBod.x, ObBod.y);
  else PopupMenu2->Popup(ObBod.x, ObBod.y);
}
```

V obsluze je nejprve zjištěno, zda bylo stisknuto pravé tlačítko myši, dále jsou uživatelské souřadnice místa kliknutí převedeny na obrazovkové souřadnice (pomocí metody *ClientToScreen*), počítán počet kliknutí (deklarujeme **PocetKliknuti** jako soukromou položku formuláře; je typu int) a zobrazíme příslušnou nabídku. Obsluhy voleb nabídky vytvořte sami. U druhé nabídky se pokuste odškrtnout aktuálně nastavenou volbu.

39. V další aplikaci se seznámíme s používáním komponenty **MaskEdit**. Jedná se o editační komponentu, která má speciální vlastnost **EditMask**. Jestliže otevřeme Editor vstupní masky (stiskem tlačítka (…) v Inspektoru objektů), můžeme snadno definovat řetězec vstupní masky, který určuje zapisovaný řetězec. Mimo zadání masky umožňuje Editor vstupní masky zadat znak použitý pro určení místa pro vstup a rozhodnout, zda se budou s řetězcem ukládat i další znaky obsažené v masce. Např. u kódu země v telefonním čísle si můžeme závorky nastavit pouze jako vstupní pomůcku nebo je ukládat i s řetězcem telefonního čísla. Tyto dva údaje odpovídají poslední části masky (implicitní oddělovač je středník). Stiskneme-li v Editoru vstupní masky tlačítko **Masks**, pak můžeme otevřít maskový soubor. Předdefinované soubory obsahují standardní masky pro některé země.

Začneme s vývojem nové aplikace. Na formulář umístíme komponentu **MaskEdit** a v Editoru vstupní masky vybereme některou masku. Před tuto komponentu umístíme text *Editace s maskou:*. Pod **MaskEdit** umístíme komponentu **Edit** a před ní text *Editační maska:*. Do spodní části formuláře umístíme komponentu **ListBox** obsahující popis znaků masky. Zadáme zde tyto řetězce znaků (komponentu zvětšíme):

```
(l) možné písmeno (letter)
(L) vyžadované písmeno
(a) možný alfanumerický znak
(A) vyžadovaný alfanumerický znak
```

```
(c) možný znak (character)
(C) vyžadovaný znak
(9) možná číslice
(0) vyžadovaná číslice
(#) číslice nebo znaménko
(_) mezera (můžeme měnit znak reprezentující mezeru)
(\) následující znak je interpretován jako literál
(>) převod na velká písmena
(<) převod na malá písmena
(<>) bez úpravy velikosti písmen
(!) potlačení úvodních mezer
(/) standardní systémový oddělovač měsíců, dnů a roků v datumu
(:) standardní systémový oddělovač hodin, minut a sekund v čase
(;) oddělovač polí masky
```

Vytvoříme ještě dvě obsluhy událostí. Obsluha události **OnCreate** formuláře bude tvořena příkazem:

```
Edit1->Text = MaskEdit1->EditMask;
```

a obsluha **OnChange** editačního ovladače příkazem:

```
MaskEdit1->EditMask := Edit1->Text;
```

Nyní již aplikaci můžeme vyzkoušet. Do editačního ovladače zadáme nějakou masku a vyzkoušíme jaký má vliv na vstup v komponentě **MaskEdit**.

40. Každý formulář může být implementovaný jako vícedokumentové rozhraní (MDI) nebo jednodokumentové rozhraní (SDI). V aplikaci MDI může být otevřeno více dokumentů nebo podřízených oken v jednom nadřízeném okně. V aplikaci SDI může být pouze jedno okno dokumentu. Typ aplikace určuje vlastnost formuláře **FormStyle**.

    Použití aplikace MDI si ukážeme na následujícím příkladu: Otevřeme nový prázdný projekt a přidáme k němu nový prázdný formulář. Nastavíme vlastnost **FormStyle** formuláře **Form1** na *fsMDIForm* a formuláře **Form2** na *fsMDIChild*. Na **Form1** přidáme komponentu **Panel** a změníme její vlastnost **Align** na *alTop*. Na panel přidáme komponentu **SpeedButton** a generujeme pro ní tuto obsluhu události **OnClick**:

    ```
    Form2->Show();
    ```

    Do *Unit1* přidáme hlavičkový soubor *Unit2***.** Spustíme aplikaci a stiskneme **SpeedButton**. **Form2** se otevře uvnitř **Form1**. Můžeme měnit jeho velikost, přesouvat jej, ale pouze v rozsahu **Form1**.

41. Předchozí aplikace nás uvedla do problematiky aplikací MDI (nebylo zde vhodné otevírat více podřízených oken). Následuje již skutečná aplikace MDI. Začneme novou aplikací. Pro formulář vytvoříme nabídku:

    ```
    Okno
    Nové
    ```

    a nastavíme tyto vlastnosti formuláře: **Caption** na *MDI rám*, **FormStyle** na *fsMDIForm* a **WindowMenu** na *Okno1*. K aplikaci přidáme další formulář, kde nastavíme: **Caption** na *Podřízené okno*, **FormStyle** na *fsMDIChild* a **Position** na *poDefaut*. Do deklarace třídy hlavního formuláře přidáme soukromou položku:

    ```
    int Citac;
    ```

    Obsluha volby **Nové** bude tvořena příkazy (deklaraci instance formuláře **Form2** můžeme v jednotce druhého formuláře vynechat a vyřadíme **Form2** z automaticky vytvářených formulářů):

    ```
    TForm2 *Form2;
    WindowMenu = Okno1;
    Citac++;
    Form2 = new TForm2(this);
    Form2->Caption = Form2->Caption + ' ' + IntToStr(Citac);
    Form2->Show();
    ```

    Po spuštění aplikace můžeme otevřít několik podřízených oken. Povšimněte si, že se jejich seznam objevuje v nabídce **Okno** (je to určeno vlastností **WindowMenu**). Zjistěte, co se stane, otevřeme-li jich více než 9. Jestliže některé z podřízených oken zavřeme, pak v seznamu oken stále existuje (není zrušeno). Toto vyřešíme v další verzi aplikace.

42. Většina MDI aplikací obsahuje v nabídce **Okno** volby pro kaskádovité a dlaždicovité uspořádání otevřených podřízených oken. Pro obsluhu těchto voleb můžeme použít některé vlastnosti a metody, které jsou u formuláře s vlastností **FormStyle** nastavenou na *fsMDIForm*. Jsou to: metoda **Cascade** (kaskádovité řazení), metoda **Tile** (dlaždicovité řazení), vlastnost **TileMode** (určuje jak bude pracovat **Tile**) a metoda **ArangeIcon** (uspořádání ikon minimalizovaných oken). Toto jsou metody a vlastnosti odvozené od Windows. Builder definuje některé další: vlastnost **ActiveMDIChild** (určuje aktivní okno; tato vlastnost je určena pouze pro čtení), metoda **Next** (aktivuje následující okno), metoda **Previous** (aktivuje předchozí okno),

vlastnost **MDIChildCount** (současný počet podřízených oken) a vlastnost **MDIChildren** (pole podřízených oken; pomocí cyklu můžeme procházet všemi podřízenými okny).

Na závěr předchozího zadání jsme se zmínili o nedostatku (nezrušení uzavřeného podřízeného okna). Tuto závadu odstraníme vytvořením obsluhy události **OnClose** podřízeného formuláře s příkazem:

```
Action = caFree;
```

Toto je jediná metoda podřízeného okna. Rám MDI aplikace nyní vybavíme následující nabídkou:

```
Soubor          Okno
Nové            Kaskáda
----            Dlaždice
Konec           Uspořádat ikony
```

Obsluhy voleb v nabídce již vytvořte sami.

43. Pro MDI aplikace (např. textový editor) potřebuje hlavní nabídka získávat prvky nabídky z jiných (podřízených formulářů). Tomuto procesu říkáme *slučování nabídek*. Při práci s nabídkami používáme vlastnost **Menu** formuláře a vlastnosti **GroupIndex** prvků nabídky. Vlastnost **Menu** určuje aktivní nabídku formuláře (můžeme ji měnit i za běhu aplikace). Vlastnost **GroupIndex** určuje pořadí, ve kterém slučujeme prvky nabídky. Slučované prvky mohou nahradit prvky na hlavním řádku nabídky (stejná hodnota **GroupIndex**) nebo mohou být vloženy.

    Slučování nabídek si ukážeme na jednoduché aplikaci (nejedná se o MDI aplikaci). Hlavní formulář bude obsahovat nabídku (v závorkách jsou uvedeny hodnoty vlastností **GroupIndex**):

```
Soubor (0)      E-Mail (1)              Nápověda (2)
Nový            Zobraz modálně          O aplikaci
Otevřít         Zobraz nemodálně
Uložit          Zavři nemodální
-----
Konec
```

    Volbu **Zavři nemodální** zakážeme. K aplikaci přidáme další formulář, který bude mít nabídku:

```
E-Mail Soubor (1)
Otevřít
Uložit
------
Zavřít
```

    Abychom si ukázali, že tento přístup funguje s modálním i nemodálním formulářem, jsou v nabídce **E-Mail** volby pro zobrazení okna oběma způsoby. Obsluha volby **Zobraz modálně** je tvořena příkazem:

```
Form2->ShowModal();
```

    Formuláře tohoto programu jsou vytvořeny při spuštění aplikace. Když program zobrazí druhý formulář jako modální, pak není možno používat nabídku hlavního formuláře. Pokud formulář zobrazíme nemodálně, měli bychom zakázat ručně volby **Zobraz modálně** a **Zobraz nemodálně** (v tomto případě není vhodné umožnit vícenásobné zobrazení formuláře). Obsluha volby **Zobraz nemodálně** tedy bude tvořena příkazy:

```
Zobrazmodln1->Enabled = false;
Zobraznemodln1->Enabled = false;
Zavinemodln1->Enabled = true;
Form2->Show();
```

    Poslední volba v nabídce **E-Mail** provádí zavření nemodálního formuláře. Obsluha bude tvořena příkazem:

```
Form2->Close();
```

    Po uzavření nemodálního formuláře, bychom ale měli vrátit nabídku do původního stavu. Formulář lze ale uzavřít mnoha způsoby. Místo opakování kódu pro povolení voleb nabídky na mnoha místech je můžeme napsat pouze do obsluhy události **OnClose** pro druhý formulář:

```
Form1->Zobrazmodln1->Enabled = true;
Form1->Zobraznemodln1->Enabled = true;
Form1->Zavinemodln1->Enabled = false;
```

    Nyní se ale programové jednotky obou formulářů odkazují na sebe navzájem. Do jednotky druhého formuláře musíme tedy přidat vložení hlavičkového souboru prvního formuláře. Pro druhý formulář vytvoříme ještě obsluhu volby **Zavřít** s příkazem *Close();*. Nyní již aplikaci můžeme vyzkoušet (zatím se ještě nejedná o slučování nabídek). Na tomto příkladě vidíme, že obsluha modálních formulářů je jednodušší než nemodálních formulářů.

    Aby naše aplikace měla nějaký význam, umístěte na hlavní formulář komponentu **ListBox** s telefonními čísly svých známých a na druhý formulář také **ListBox**, tentokrát s E-Mail adresami. Vytvořte také ostatní obsluhy voleb v nabídce (ukládání seznamů do souborů, otevírání souboru atd.).

44. V předchozí aplikaci měli oba formuláře vlastní nabídky. Builder ale podporuje slučování nabídek. Je to prováděno takto. Hlavní formulář má stále nabídku. Ostatní formuláře mají nabídky s nastavenou vlastností **AutoMerge** na *True* (jejich nabídka potom nebude zobrazována ve formuláři, ale bude spojena s hlavní nabídkou). Slučování proběhne podle hodnot vlastností **GroupIndex**. V našem příkladě nabídka **E-Mail Soubor** nahradí nabídku **E-Mail** (obě mají stejnou hodnotu **GroupIndex**). Slučování nabídek má význam pouze pro nemodální formuláře (u modálních formulářů se k nabídce nedostaneme, takže Builder slučování neprovádí). Po uzavření nemodálního formuláře se ale neobnoví původní hlavní nabídka (formulář není zrušen, pouze se skryl). Problém vyřešíme zásadní změnou přístupu k obsluze druhého formuláře. Místo vytváření objektu formuláře při spouštění aplikace jej musíme vytvořit a zrušit pokaždé, když má být zobrazen nebo skryt. V dialogovém okně **Project Options** na stránce **Forms** zrušíme automatické vytváření druhého formuláře. Abychom mohli formulář zobrazit jako modální, musíme jej vytvořit, nastavit jeho vlastnosti a později jej zrušit. Obsluha volby **Zobraz modálně** bude tedy tvořena příkazy:

```
Form2 = new TForm2(self);
Form2->MainMenu1->AutoMerge = false;
Form2->ShowModal();
Form2->Free();
```

Pro **Zobraz nemodálně** to bude:

```
Form2 = new TForm2(self);
Form2->MainMenu1->AutoMerge = true;
Form2->Show();
```

Při této volbě formulář zrušen není (není zde již nutné zakazovat volby v nabídce). Aby při uzavření formuláře byl formulář opravdu zrušen, musíme ještě u události **OnClose** druhého formuláře nastavit hodnotu parametru **Action** na *caFree*. Nyní je již aplikace hotova.

45. Zatím vytvořené MDI aplikace neprováděli nic rozumného. Obdrželi jsme program s typickým chováním MDI, ale s žádným praktickým užitkem. S podřízeným formulářem můžeme ale provádět cokoliv. Můžeme přidat různé komponenty, vytvořit editory apod. Kterýkoli z programů, které jsme zatím vytvořili můžeme převést na MDI aplikaci (u některých to ale nedává smysl). Naší skutečnou první MDI aplikací bude jednoduchá verze grafického programu. Tento program na místě kliknutí myší zobrazí kruh. Podřízené okno obsahuje tuto nabídku (v aplikaci již použijeme slučování nabídek):

```
Kruh
Barva výplně ...
Barva okraje ...
Šíře okraje ...
---------------
Získej pozici
```

Důležité je to, že při programování podřízeného formuláře nemusíme brát v úvahu existenci jiných formulářů (včetně formuláře rámu). Jediný speciální případ je slučování nabídek. Vezmeme rám ze druhé MDI aplikace s jedinou změnou, kterou v něm provedeme je nastavení vlastností **GroupIndex** pro nabídku **Soubor** na 1 a pro nabídku **Okno** na 3. To je vše, co na tomto formuláři změníme. **GroupIndex** u nabídky **Kruh** v podřízeném formuláři nastavíme na 2. U podřízeného formuláře nastavíme tyto vlastnosti: **Caption** na *Podřízené okno*, **Color** na *clTeal*, **FormStyle** na *fsMdiChild*, **Menu** na *MainMenu1* a **Position** na *poDefault*. Do soukromé části podřízeného formuláře přidáme deklarace:

```
int XStred, YStred;
int SireOkraje;
TColor BarvaOkraje, BarvaVyplne;
```

Nyní již zbývá vytvořit několik obsluh událostí. Obsluha **OnCreate** formuláře je tvořena příkazy:

```
XStred = -200;
YStred = -200;
SireOkraje = 1;
BarvaOkraje = clBlack;
BarvaVyplne = clYellow;
```

Obsluhu **OnPaint** vytvoříme příkazy:

```
Canvas->Pen->Width = SireOkraje;
Canvas->Pen->Color = BarvaOkraje;
Canvas->Brush->Color = BarvaVyplne;
Canvas->Ellipse(XStred-30, YStred-30, XStred+30, YStred+30);
```

Obsluhu **OnMouseDown** tvoří příkazy:

```
XStred = X;
YStred = Y;
Repaint();
```

Obsluha **OnClose** je stejná jako v předcházející MDI aplikaci. Zbývá ještě vytvořit několik obsluh voleb v nabídce. Např. obsluhu volby **Šíře okraje** tvoří příkazy:

```
AnsiString VstupniRetezec = IntToStr(SireOkraje);
if (InputQuery("Okraj", "Zadej šířku", VstupniRetezec)){
  SireOkraje = VstupniRetezec.ToIntDef(SireOkraje);
  Repaint();
}
```

Ostatní obsluhy voleb vytvořte sami. Výpis informací volbou **Získej pozici** provádějte funkcí *ShowMessage*. Aplikaci můžeme vyzkoušet. Každý podřízený formulář může obsahovat pouze jeden kruh. Podřízených formulářů můžeme otevřít více.

46. Obecně mohou MDI aplikace obsahovat více druhů podřízených formulářů. Ukážeme si to na rozšíření předchozí aplikace. Budeme muset vytvořit nový podřízený formulář. Tentokrát vytvoříme formulář s pohybujícím se čtvercem, který se bude odrážet od okrajů. K předchozí aplikaci přidáme další formulář, přidáme k němu následující nabídku:

```
Čtverec               Pohyb
Barva výplně ...      Start
--------------        Stop
Získej pozici
```

U obou těchto nabídek nastavíme **GroupIndex** na 2 a volbu **Start** zakážeme. U formuláře nastavíme tyto vlastnosti: **AutoScroll** na *False*, **Caption** na *Pohybující se čtverec*, **Color** na *clAqua*, **FormStyle** na *fsMDIChild*, **Menu** na *MainMenu1*, **Position** na *poDefault* a **Visible** na *True*. Na formulář vložíme dále komponentu **Shape** a nastavíme pro ní vlastnosti: **Left** na 40, **Top** na 48, **Width** a **Height** na 30, **Brush.Pen** na *clFuchsia*, **Pen.Color** na *clBlue*, **Pen.Width** na 2 a **Shape** na *stSquare*. Na formulář přidáme ještě komponentu **Timer** a nastavíme jeho vlastnost **Interval** na 200. Nyní se již můžeme zabývat pohybem čtverce. Nejprve nadefinujeme výčtový typ **Smery** s možnými směry pohybu čtverce. Deklaraci tohoto typu umístíme do hlavičkového souboru před deklaraci typu formuláře:

```
enum Smery {nahoru_vpravo, dolu_vpravo, dolu_vlevo, nahoru_vlevo};
```

Jako soukromou položku třídy formuláře přidáme:

```
Smery Smer;
```

Obsluhu **OnCreate** bude tvořit příkaz:

```
Smer = dolu_vlevo;
```

a obsluha **OnClose** je tvořena obvyklým příkazem. Obsluhu volby nabídky **Start** tvoří příkazy:

```
Timer1->Enabled = true;
Start1->Enabled = false;
Stop1->Enabled = true;
```

Obsluhy ostatních voleb v nabídce vytvořte sami. Zbývá ještě vytvořit obsluhu události **OnTimer** časovače. V této obsluze budeme měnit souřadnice čtverce. Začátek obsluhy je tvořen příkazy:

```
switch (Smer){
  case nahoru_vpravo:
    Shape1->Left += 3;
    Shape1->Top -= 3;
    if (Shape1->Top <= 0) Smer = dolu_vpravo;
    if (Shape1->Left+Shape1->Width>=ClientWidth) Smer = nahoru_vlevo;
    break;
  case nahoru_vlevo:
  ...
}
```

Zbytek obsluhy vytvořte sami. Tím je hotov druhý podřízený formulář. Musíme ještě změnit hlavní formulář. Změníme zde nabídku:

```
Soubor                Okno
Nový kruh             Kaskáda
Nový čtverec          Dlaždice vodorovně
Uzavřít vše           Dlaždice svisle
--------------        Uspořádat ikony
Konec                 Počet
```

V nabídce **Soubor** si ukážeme obsluhu volby **Uzavřít vše** (ostatní vytvořte sami):

```
for (int I = MDIChildCount - 1; I >= 0; I--) MDIChildren[I]->Close();
```

Volba **Dlaždice vodorovně** bude obsloužena příkazy:

```
TileMode = tbHorizontal;
Tile();
```

Obdobně vytvořte obsluhu volby **Dlaždice svisle** (*tbVertical*). Obsluhu volby **Počet** tvoří příkazy:

```
int NCtverec = 0, NKruh = 0;
for (int I = 0; I < MDIChildCount; I++)
  if (dynamic_cast<TForm3 *> (MDIChildren[I])) NCtverec++;
  else NKruh++;
ShowMessage("Jsou zde "+IntToStr(MDIChildCount)+" podřízené formuláře.\n"
            +IntToStr(NKruh)+" je podřízených oken kruhů a \n" +
            IntToStr(NCtverec) + " podřízených oken čtverců.");
```

Ostatní obsluhy vytvořte sami a aplikaci vyzkoušejte.

47. Řada aplikací bývá také vybavena paletou nástrojů. Paleta nástrojů obvykle obsahuje určitý počet tlačítek, která můžeme stisknout myší a které spouští příkazy nebo přepínají volby. Paleta nástrojů může také někdy obsahovat komponentu **ComboBox**, editační ovladač nebo jiný ovládací prvek. Poslední generace palet nástrojů mohou být přemisťovány po ploše okna nebo mohou být umístěny úplně mimo jako samostatné okno s polem tlačítek. Aplikace může také obsahovat stavový řádek, který má obvykle jednu nebo více ploch s textovým popisem současného stavu programu. Může zde být plocha pro souřadnice, pro zobrazení aktivního písma nebo pro zobrazení nápovědy, chybového hlášení atd. Pro vytvoření palety nástrojů nebo stavového řádku můžeme použít komponentu **Panel** a do ní můžeme přidat tlačítka nebo další panely. Panel můžeme považovat za nástroj pro rozdělení uživatelské plochy na různé části a pro seskupení dalších komponent. Panel může mít vlastní text, ale většinou se tato možnost nevyužívá. Panely ale často vytvářejí trojrozměrné zobrazení (pomocí vlastností **BevelInner** a **BevelOuter**) a zlepšují tak vzhled aplikace. Panel obvykle umisťujeme na určitou plochu formuláře a měníme hodnotu jeho vlastnosti **Align**. U typické palety nástrojů použijeme hodnotu *alTop* a u stavového řádku použijeme *alBottom*. Panel může využívat vlastnosti **Hint** a **ShowHints** k zobrazování bublinkové nápovědy.

    V následující aplikaci se pokusíme vytvořit typický panel nástrojů. Panel umístíme na horní okraj formuláře a na něj vložíme několik komponent **SpeedButton**. **SpeedButton** se podobá **BitBtn** a může mít také titulek a symbol, ačkoliv se používají pouze grafické elementy. Komponenty **SpeedButton** se mohou chovat jako normální tlačítka, přepínací tlačítka nebo značky a mohou mít různé bitové mapy pro různé situace.

    Začneme s vývojem nové aplikace. Na formulář umístíme komponentu **Panel** a nastavíme její vlastnost **Align** na *alTop*. Na formulář dále umístíme komponentu **Label**, u které nastavíme vlastnosti: **Align** na *alClient*, **AutoSize** na *False*, **Caption** na *Text, na kterém budeme ukazovat vliv tlačítek na paletě komponent*, **ParentFont** na *False* a **WordWrap** na *True*. U formuláře nastavíme vlastnost **ActiveControl** na *Panel1*. Na panel budeme postupně přidávat komponenty **SpeedButton** podle následujícího obrázku (symboly tlačítek se pokusíme najít v knihovně obrázků nebo je nakreslíme; rozměry 17 x 17).

    Jestliže přidáme **SpeedButton** můžeme jednoduše napsat jeho obsluhu události **OnClick** (implicitní událost). V našem programu k prvnímu tlačítku přiřadíme příkaz:

```
MessageBeep($FFFF);
```

    Dále přidáme skupinu tlačítek, která budou fungovat jako přepínací tlačítka. U všech nastavíme hodnotu **GroupIndex** na stejnou hodnotu (v našem případě na 1). V naši aplikaci budou tři a budeme jimi ovládat zarovnávání textu. Jedno z těchto tlačítek by mělo být vybráno (nastavíme jeho vlastnost **Down** na *True*; v našem případě to bude tlačítko označující zarovnávání textu doleva). Obsluhy stisku těchto tlačítek budou obdobné. Pro tlačítko zarovnávání vlevo bude použit příkaz:

```
Label1->Alignment = taLeftJustify;
```

    Vytvořte obsluhy pro stisk dalších dvou tlačítek. Další tři tlačítka budou určovat styl použitého textu. Může jich být stisknuto několik (nebo i žádné). Pro všechna nastavíme **GroupIndex** na stejnou hodnotu (v našem případě na 2) a **AllowAllUp** na *True*. Pro všechna tři tlačítka zde bude stejná obsluha **OnClick** a bude tvořena příkazy:

```
TFontStyles X;
if (SpeedButton5->Down) X << fsBold;
if (SpeedButton6->Down) X << fsItalic;
if (SpeedButton7->Down) X << fsUnderline;
Label1->Font->Style = X;
```

    Poslední tlačítko může zůstat stisknuté. To dosáhneme tím, že jeho hodnotu **GroupIndex** nastavíme na jinou hodnotu než mají ostatní tlačítka (např. na 3) a **AllowAllUp** na *True*. Obsluha **OnClick** tohoto tlačítka bude tvořena příkazy:

```
if (SpeedButton8->Down) Label1->Font->Size = 24;
else Label1->Font->Size = 12;
```

Aplikace je hotova a můžeme ji vyzkoušet.

48. Naše aplikace z předchozího zadání má standardní chování. Každý **SpeedButton** může využívat i více různých bitových map bez toho, aby je bylo nutno měnit ručně. Je to obdoba **BitBtn** z aplikace vytvořené podle zadání 15 (tlačítko **Pal**). V následujícím rozšíření předchozí aplikace si také ukážeme jak některá tlačítka palety nástrojů zakázat. Do předchozí aplikace přidáme tuto nabídku:

```
Soubor          Paleta nástrojů      Nápověda
Konec           Viditelná            O aplikaci
                Zakázat zvuk
                Zakázat styly
                Zakázat velikost
```

Vlastnost **Checked** u prvku nabídky **Viditelná** nastavíme na *True*. Můžeme začít s vytvářením obsluh jednotlivých voleb. Volba **Viditelná** bude obsloužena příkazy:

```
Panel1->Visible = ! Panel1->Visible;
Viditeln1->Checked = ! Viditeln1->Checked;
```

Obdobně pro volbu **Zakázat zvuk** změníme hodnotu dvou vlastností typu Bool (**SpeedButton1->Enabled** a **Zakzatzvuk->Checked**). U volby **Zakázat styly** musíme povolit nebo zakázat všechna tři tlačítka stylů. Vytvořte zbývající obsluhy sami.

Normálně Builder používá pro **SpeedButton** tři verze bitových map: normální, zakázané a stisknuté. Pro poslední tlačítko vytvoříme vlastní verzi symbolů (72 x 17):

Poslední stav je zůstaň stisknutý. Aplikaci můžeme vyzkoušet. Je zde ale jeden nedostatek. Jestliže máme stisknuté některé z tlačítek stylů a tlačítka stylů zakážeme a opět povolíme, pak stisknuté tlačítko je překresleno jako uvolněné. Problém je v přechodu mezi stavy zakázáno a stisknuto. Problém můžeme vyřešit vynucením obnovení symbolu tlačítka jeho uvolněním a stisknutím, např.

```
SpeedButton5->Down = false;
SpeedButton5->Down = true;
```

49. Další prvek, který se v paletách nástrojů používá je bublinková nápověda. Její používání zajistíme změnou některých vlastností. U komponenty **Panel** nastavíme vlastnost **ShowHint** na *True* a **ParentShowHint** na *False*. U jednotlivých tlačítek nastavíme vlastnosti **Hint**. Např. u posledního tlačítka použijeme text *Zvětšit*. Volbu textů ostatních bublinkových nápověd proveďte sami. U posledního tlačítka by bylo vhodné měnit text nápovědy podle stavu tlačítka. Změníme tedy obsluhu stisku tohoto tlačítka:

```
if (SpeedButton8->Down){
  Label1->Font->Size = 24;
  SpeedButton8->Hint = "Zmenšit";}
else {
  Label1->Font->Size = 12;
  SpeedButton8->Hint = "Zvětšit";}
```

Při přidávání nápovědy není třeba téměř žádné kódování. Kód je zapotřebí pouze u nápověd závislých na kontextu.

50. Do palety nástrojů můžeme přidat i další ovladače. Ukážeme si to v dalším rozšíření předchozí aplikace. Zde nejprve zrušíme poslední **SpeedButton**, obsluhu stisku tohoto tlačítka a volbu v nabídce. Na panel přidáme komponentu **ComboBox** a budeme s ní provádět volbu použitého písma. Vlevo od této komponenty přidáme komponentu **Label** s textem *Písmo:* a nastavíme její vlastnost **Hint** na *Vyber písmo*. U komponenty **ComboBox** nastavíme vlastnosti: **Hint** na *Vyber písmo* a **Style** na *csDropDownList*. Pro formulář musíme vytvořit obsluhu události **OnCreate**, ve které do kombinovaného ovladače vložíme seznam dostupných písem:

```
for (int I = 0; I < Screen->Fonts->Count; I++)
  ComboBox1->Items->Add(Screen->Fonts->Strings[I]);
ComboBox1->ItemIndex = ComboBox1->Items->IndexOf(Label1->Font->Name);
```

Dále musíme vytvořit obsluhu **OnChange** kombinovaného ovladače. Bude tvořena příkazem:

```
Label1->Font->Name = ComboBox1->Items->Strings[ComboBox1->ItemIndex];
```

To je vše, co se týká této komponenty. Pro komponentu **Panel** přidáme ještě místní nabídku:

```
Skrýt paletu nástrojů
---------------------
Zakázat zvuk
Zakázat styly
```

Pro panel musíme tedy nastavit **PopupMenu** na **PopupMenu1**, pro nabídku vytvořit obsluhu **OnPopup** (překopírujeme označení prvků hlavní nabídky do místní nabídky) s příkazy:

```
Zakzatzvuk2->Checked = Zakzatzvuk1->Checked;
Zakzatstyly2->Checked = Zakzatstyly1->Checked;
```

Zbytek potřebných věcí pro používání místní nabídky dokončete sami.

51. Některé aplikace umožňují přemisťování palety nástrojů. Vyjdeme z předposlední aplikace s paletou nástrojů (ještě před použitím kombinovaného ovladače) a změníme ji tak, že panel nástrojů budeme moci přetáhnout myší na některý jiný okraj formuláře. U komponenty **Panel** změníme vlastnost **DragMode** na *dmAutomatic*. V naši aplikaci využijeme standardní výčtový typ **TAlign** a k deklaraci třídy formuláře přidáme tyto soukromé položky a metody:

```
TAlign PozicePalety;
int SirePanelu;
TAlign ZiskejPoziciPalety(int X, int y);
void OtocPaletu(void);
```

V jednotce formuláře deklarujeme také konstantu určující šířku okraje formuláře, kde je akceptováno tažení:

```
const int SireOkraje = 20;
```

Metoda **ZiskejPoziciPalety** vrací novou pozici palety na základě předaných souřadnic. Tato metoda vypadá takto:

```
TAlign TForm1::ZiskejPoziciPalety(int X, int Y){
  if (X < SireOkraje) return alLeft;
  if (Y < SireOkraje) return alTop;
  if (X > ClientWidth - SireOkraje) return alRight;
  if (Y > ClientHeight - SireOkraje) return alBottom;
  return alNone;
}
```

Metoda **OtocPaletu** provede změnu z vodorovné na svislou paletu nebo naopak. Provádí se zde pouze záměna vlastností **Top** a **Left**:

```
void TForm1::OtocPaletu(void){
  for (int I = 0; I < Panel1->ControlCount; I++){
    int X = Panel1->Controls[I]->Top;
    int Y = Panel1->Controls[I]->Left;
    Panel1->Controls[I]->Top = Y;
    Panel1->Controls[I]->Left = X;
  }
}
```

Obsluha **OnCreate** formuláře je tvořena příkazy:

```
PozicePalety = alTop;
SirePanelu = Panel1->Height;
```

Nyní již zbývá vytvořit obsluhy událostí **OnDragOver** a **OnDragDrop** pro komponentu **Label**. Obsluha **OnDragOver** je tvořena příkazem:

```
if(ZiskejPoziciPalety(X+Label1->Left,Y+Label1->Top)==alNone)Accept=false;
else Accept = true;
```

a obsluha **OnDragDrop** příkazy:

```
TAlign PP;
PP = ZiskejPoziciPalety(X+Label1->Left,Y+Label1->Top);
Panel1->Align = PP;
if ((((PP == alTop) || (PP == alBottom)) &&
   ((PozicePalety == alLeft) || (PozicePalety == alRight))) ||
   (((PP == alLeft) || (PP == alRight)) &&
   ((PozicePalety == alTop) || (PozicePalety == alBottom))))
     OtocPaletu();
PozicePalety = PP;
```

Nyní již aplikaci můžeme vyzkoušet.

52. V další aplikaci si ukážeme, jak na stavovém řádku zobrazovat nápovědu k nabídce. Začneme s vývojem nové aplikace. V této aplikaci vytvoříme nabídku (nezáleží na jejím obsahu). Do vlastností **Hint** jednotlivých prvků nabídky přiřadíme texty, které budeme chtít zobrazovat na stavovém řádku. Dále musíme napsat obsluhu události **OnHint** u aplikace. Tuto obsluhu musíme do formuláře přidat ručně a potom např. v obsluze **OnCreate** formuláře ji přiřadit **OnHint** objektu aplikace. Do deklarace třídy formuláře přidáme veřejnou metodu:

```
void __fastcall ShowHint(TObject *Sender);
```

a v obsluze **OnCreate** formuláře bude příkaz:

```
Application->OnHint = ShowHint;
```

Na formulář přidáme komponentu **Panel** a nastavíme pro ní tyto vlastnosti: **Align** na *alBottom*, **Alignment** na *taLeftJustify* a **BevelInner** na *bvLowered*. Zbývá ještě vytvořit metodu **ShowHint**:

```
void __fastcall TForm1::ShowHint(TObject *Sender){
  Panel1->Caption = Application->Hint;
}
```

Nyní již můžeme vyzkoušet zobrazování nápovědy k nabídce na stavovém řádku.

6. Práce se seznamy

1. Mnoho různých aplikací Builderu potřebuje pracovat se seznamem znakových řetězců. Tyto seznamy jsou prvky v okně seznamu nebo kombinovaného okna, řádky textu v komponentě **Memo**, seznam podporovaných písem, řádky a sloupce v mřížce řetězců. Přestože aplikace používají tyto seznamy různým způsobem, Builder poskytuje společné rozhraní prostřednictvím objektu nazvaného *seznam řetězců*. Seznam řetězců používáme i prostřednictvím Inspektora objektů. Vlastnost seznamu řetězců se zde zobrazuje hodnotou *TStrings* ve sloupci hodnot. Když zde dvojitě klikneme je zobrazen Editor seznamu řetězců, ve kterém můžeme editovat, přidávat nebo rušit řádky.

   K zjištění počtu řetězců v seznamu používáme vlastnost **Count**. Je to vlastnost určená pouze pro čtení. Jelikož indexy používané v seznamu řetězců začínají od nuly, je hodnota **Count** o jedničku větší než index posledního řetězce v seznamu. Např. aplikace může zjistit, kolik různých písem je aktuálně podporováno obrazovkou, přečtením vlastnosti **Count** objektu seznamu písem (seznam jmen všech podporovaných písem), tj. *Screen->Fonts->Count*. Seznam řetězců má indexovanou vlastnost nazvanou **Strings**, kterou si můžeme představit jako seznam řetězců. Např. první řetězec v seznamu je *Strings[0]*. Řetězce do seznamu lze přidávat dvěma způsoby. Jedná se o přidání na konec seznamu a vložení do seznamu na určený index. K přidání řetězce na konec seznamu voláme metodu seznamu *Add*, přidávající parametr jako nový řetězec. K vložení řetězce do seznamu voláme metodu *Insert* mající dva parametry: index na který chceme řetězec vložit a vkládaný řetězec.

   Vytvořte aplikaci obsahující komponentu **Memo** a tlačítko. Při stisku tlačítka zapište do komponenty **Memo** jména všech podporovaných písem obrazovkou.

2. Se seznamem řetězců můžeme provádět ještě řadu dalších operací. Často potřebujeme zjistit pozici (index) řetězce v seznamu (nebo zjistit existenci řetězce). K lokalizaci řetězce v seznamu používáme metodu seznamu *IndexOf*. Metoda přebírá řetězec jako svůj parametr a vrací index nalezeného řetězce nebo –1 není-li řetězec nalezen. Tato metoda pracuje pouze s kompletními řetězci (parametr se musí celý přesně shodovat s celým řetězcem v seznamu). Např. následující příkaz zjišťuje zda soubor WIN.INI je v seznamu souborů v okně seznamu souborů:

   ```
   if (FileListBox1->Items->IndexOf("WIN.INI") > -1) ...
   ```

   Řetězec v seznamu můžeme přesunout na jinou pozici. K přesunu řetězce v seznamu voláme metodu seznamu *Move*, která má dva parametry: současný index prvku a index, na který prvek chceme přesunout. Pro zrušení řetězce v seznamu voláme metodu seznamu *Delete* a předáme ji index rušeného řetězce. Jestliže index rušeného řetězce neznáme, použijeme metodu *IndexOf* k jeho lokalizaci. Např. příkazem, který přidáme do naší předchozí aplikace můžeme ze seznamu písem odstranit písmo *Curier New*:

   ```
   TStrings * X = Memo1->Lines;
   if (X->IndexOf("Courier New") > -1)
     X->Delete(X->IndexOf("Courier New"));
   ```

   Kopírování kompletního seznamu řetězců z jednoho seznamu do jiného, provedeme přiřazením zdrojového seznamu cílovému seznamu. Např. *Memo1->Lines := ListBox1->Items;*. Tím přepíšeme řetězce cílového seznamu. Jestliže chceme přidat seznam řetězců na konec jiného seznamu, voláme metodu *AddStrings* seznamu ke kterému přidáváme a jako parametr předáme seznam řetězců, který chceme přidat. Např. *Memo1->AddStrings(ListBox1->Items);*. Doplňte předchozí aplikaci o komponentu **ListBox**, do které při stisku dalšího tlačítka překopírujete všechny řetězce z komponenty **Memo**, které obsahují ve svém textu *CE*.

3. Pomocí cyklu lze procházet jednotlivými řetězci seznamu. Do předchozí aplikace přidejte další tlačítko, jehož stiskem provedeme převod řetězců v komponentě **Memo** na velká písmena. Pro převod řetězce na velká písmena lze použít metodu *UpperCase* třídy **AnsiString**.

4. Procvičování používání seznamů řetězců budeme provádět v aplikaci, která bude vytvářet anglické věty. Nejprve si ukážeme aplikaci bez použití seznamů. Začneme novou aplikaci. K hornímu okraji formuláře umístíme komponentu **Label**, ve které budeme vypisovat vytvářené věty. Na zbývající plochu formuláře umístíme vedle sebe tři komponenty **GroupBox** a změníme jejich **Caption** na „První objekt“, „Umístění“ a „Druhý objekt“. Na levý **GroupBox** (s titulkem *První objekt*) vložíme nad sebe 4 voliče s texty: „The book“, „The pen“, „The pencil“ a „The chair“. Prostřední **GroupBox** bude obsahovat 3 voliče s texty: „on“,

„under“ a „near“. Pravý **GroupBox** bude obsahovat voliče s texty: „the table“, „the big box“, „the carpet“ a „the computer“. Všechny tři horní voliče vybereme. U komponenty **Label** zvětšíme písmo a jeho **Caption** změníme na „The book is on the table.“. Tento text budeme ovládat voliči. Všechny voliče budou mít stejnou obsluhu události **OnClick**. Obsluha bude tvořena příkazy:

```
AnsiString Veta;
int I;
for (I = 0; I < GroupBox1->ControlCount; I++)
  if ((dynamic_cast<TRadioButton *> (GroupBox1->Controls[I]))->Checked)
    Veta=Veta+(dynamic_cast<TRadioButton*>(GroupBox1->Controls[I]))->Caption;
Veta = Veta + " is ";
for (I = 0; I < GroupBox2->ControlCount; I++)
  if ((dynamic_cast<TRadioButton *> (GroupBox2->Controls[I]))->Checked)
    Veta=Veta+(dynamic_cast<TRadioButton*>(GroupBox2->Controls[I]))->Caption;
Veta = Veta + " ";
for (I = 0; I < GroupBox3->ControlCount; I++)
  if ((dynamic_cast<TRadioButton *> (GroupBox3->Controls[I]))->Checked)
    Veta=Veta+(dynamic_cast<TRadioButton*>(GroupBox3->Controls[I]))->Caption;
Label1->Caption = Veta + ".";
```

Aplikace je hotova, můžeme ji vyzkoušet.

5. Změňte předchozí aplikaci tak, že krajní komponenty **GroupBox** nahradíte komponentami **ListBox**. Umožní nám to použití více slov. Do obou komponent vložíme tato slova: big box, book, carpet, chair, computer, desk, floor, pen, pencil, small box, sofa a table (můžeme je vložit pouze do jedné komponenty a v obsluze **OnCreate** formuláře překopírovat tento seznam do druhého seznamu). Aby se zlepšila použitelnost nastavíme u obou komponent **ListBox** vlastnost **Sorted** na *True*. Slova jsou uvedena bez členů. Doplňujte je při vytváření věty. Vybraný prvek v seznamu je určen hodnotou vlastnosti **ItemIndex**. Proveďte tuto změnu aplikace sami.
6. V aplikaci provedeme další změnu. Vybrané slovo v prvním seznamu odstraníme z druhého seznamu a slovo vybrané v druhém seznamu odstraníme z prvního seznamu. Po změně výběru v některém seznamu opět odstraněné slovo do seznamu vrátíme. Je nutno si tedy odstraněná slova zapamatovat. Do soukromé části deklarace formuláře vložíme položky (pro zapamatování odstraněných slov):

   ```
   AnsiString String1, String2;
   ```

   Další problém, který musíme vyřešit je to, že na začátku není v žádné komponentě **ListBox** vybrána položka. V obsluze **OnCreate** formuláře tedy vložíme do položky **String1**, resp. **String2** řetězec *book*, resp. *table* (odpovídá to původně zobrazené větě), tyto slova zrušíme v příslušných seznamech (**String1** v **ListBox2** a **String2** v **Listbox1**) a v seznamech je vybereme. V aplikaci dále umožníme přidávání dalších slov. Na formulář umístíme editační ovladač a tlačítko s textem *Přidej*. Po stisku tlačítka zjistíme, zda editační ovladač není prázdný nebo zda neobsahuje slovo, které je již obsaženo v některém seznamu a pokud tyto podmínky jsou splněny, pak obsah editačního ovladače vložíme do obou seznamů (po vložení musíme opět vybrat použitá slova ve větě; tzn. nalézt v každém seznamu použité slovo a vybrat je). V případě nesplnění podmínky zobrazte okno zpráv s oznámením chyby. Vytvořte tuto aplikaci sami.
7. Seznam řetězců lze snadno uložit do textového souboru (*SaveToFile*) a opět jej zavést nebo jej zavést do jiného seznamu (*LoadFromFile*). Jako parametr u těchto metod se používá jméno souboru. Upravte předchozí aplikaci tak, aby slova použitá v seznamu byla při spuštění aplikace načtena z textového souboru *slova.txt*, vytvořena kopie tohoto souboru (*slova.old*) a při uzavření aplikace (např. v obsluze události **OnDestroy** formuláře) opět zapsána do *slova.txt* (do seznamu jsme mohli nějaké slovo přidat). Vytvořte také potřebný textový soubor. Aplikaci vyzkoušejte.
8. Naši aplikaci nyní změníme takto: V prvním seznamu umožníme vícenásobný výběr (nebudeme také již odstraňovat použitá slova ze seznamů). Vícenásobný výběr umožníme nastavením vlastnosti **MultiSelect** na *True*. Vybrané prvky nyní zjistíme testováním pole **Selected**. Např. zjištění počtu vybraných položek provedeme příkazy:

   ```
   int PocetVybr = 0;
   for (int I = 0; I < ListBox1->Items->Count; I++)
     if (ListBox1->Selected[I]) PocetVybr++;
   ```

   Tuto změnu proveďte sami. Snažte se, aby vytvářené věty vypadaly takto: The book is …, The book, and the computer are …. The book, the computer, and the pen are …. Uvědomte si, že v seznamu nemusí být vybrán žádný prvek (v tomto případě vytvořte větu začínající *Nothing is …*).
9. V další aplikaci se podrobněji seznámíme s komponentou **ComboBox** (tyto komponenty zabírají méně místa než **ListBox** a jsou kombinací editačního ovladače a **ListBox**u). Vlastností **Style** můžeme měnit chování kombinovaného ovladače. Jsou zde tyto možnosti: *csDropDown* (umožňuje přímou editaci a na požádání

zobrazuje seznam), *csDropDownList* (neumožňuje editaci, pouze výběr ze seznamu), *csSimple* (umožňuje přímou editaci a seznam zobrazen není), *csOwnerDrawFixed* a *csOwnerDrawVariable* (seznamy s uživatelsky definovanými zobrazeními).

V následující aplikaci si ukážeme použití prvních tří z těchto stylů. Začneme s vývojem nové aplikace. Na formulář umístíme tři komponenty **ComboBox** a tlačítko s textem *Přidej*. Do seznamu řetězců každé komponenty **ComboBox** vložíme asi 20 jmen a nastavíme jejich vlastnosti **Sorted** na *True*. U prvního kombinovaného ovladače nastavíme vlastnost **Style** na *csDropDown*. Vytvoříme ještě obsluhu události **OnClick** tlačítka *Přidej*:

```
if ((ComboBox1->Text<>"")&(ComboBox1->Items->IndexOf(ComboBox1->Text)<0))
  ComboBox1->Items->Add(ComboBox1->Text);
```

Jestliže uživatel stiskne tlačítko, pak text zadaný do kombinovaného ovladače (je-li nějaký) je přidán do seznamu, samozřejmě za předpokladu, že tam již není.

U druhého kombinovaného ovladače nastavíme vlastnost **Style** na *csDropDownList*. Nepřiřadíme mu žádnou obsluhu události. Použijeme jej pro experimentování s automatickými vyhledávacími technikami. Stisknemeli klávesu s nějakým písmenem, bude vybrán první řetězec v seznamu, začínající tímto písmenem. Stiskem kláves se šipkou nahoru nebo dolů se můžeme v seznamu pohybovat bez nutnosti jej otevřít. Tyto postupy lze použít i u ostatních kombinovaných ovladačů.

Třetí kombinovaný ovladač (vlastnost **Style** nastavíme na *csSimple*) přidá do seznamu nový prvek při stisku klávesy Enter. Vytvoříme pro něj tuto obsluhu události **OnKeyPress** (stisknutí klávesy):

```
if (Key == '\n')
  if ((ComboBox3->Text<>"")&(ComboBox3->Items->IndexOf(ComboBox3->Text)<0))
    ComboBox3->Items->Add(ComboBox3->Text);
```

Tím je vývoj této aplikace dokončen. Vyzkoušíme chování jednotlivých kombinovaných ovladačů.

10. Často seznam řetězců používáme jako část nějaké komponenty a nemusíme tedy vytvářet seznam sami. Můžeme ale také vytvořit seznam řetězců, který není přiřazen žádné komponentě. Pokud vytvoříme svůj vlastní seznam, nesmíme zapomenout po ukončení práce s ním, jej opět uvolnit. Existují dva různé způsoby používání seznamu: seznam, který aplikace vytvoří, použije a zruší v jedné metodě a seznam, který aplikace vytváří, používá při běhu a zruší jej před svým ukončením.

    Jestliže seznam řetězců potřebujeme kdykoliv při běhu aplikace, je nutno jej vytvořit ihned po spuštění aplikace a uvolnit jej před ukončením aplikace. To vyřešíme takto: Do třídy hlavního formuláře aplikace přidáme položku typu **TStringList**, vytvoříme obsluhu pro událost **OnCreate** hlavního formuláře, v této obsluze vytvoříme objekt seznamu řetězců a v obsluze události **OnDestroy** hlavního formuláře (tato událost je provedena po odstranění formuláře z obrazovky před ukončením aplikace) seznam řetězců zrušíme.

    V následující aplikaci se pokusíme vytvořit seznam se souřadnicemi kliknutí myši a na závěr aplikace tento seznam zapíšeme do souboru. Začneme vývojem nové aplikace. Formulář zůstane prázdný. Jako veřejnou položku formuláře přidáme:

    ```
    TStringList *SeznamKliknuti;
    ```

    Obsluha události **OnCreate** formuláře bude tvořena příkazem:

    ```
    SeznamKliknuti = new TStringList();
    ```

    Obsluha **OnMouseDown** bude přidávat informace o kliknutí do seznamu:

    ```
    SeznamKliknuti->Add("Kliknutí na " + IntToStr(X) + ", " + IntToStr(Y));
    ```

    Zbývá ještě vytvořit obsluhu **OnDestroy**. Zde zapíšeme seznam do souboru. Jméno tohoto souboru bude vytvořeno ze jména aplikace, ve kterém příponu změníme na LOG (pokud nepřejmenujete aplikaci, pak to bude soubor *PROJECT1.LOG*). V obsluze také seznam uvolníme:

    ```
    SeznamKliknuti->SaveToFile(ChangeFileExt(Application->ExeName, ".LOG"));
    SeznamKliknuti->Free();
    ```

    Tím je aplikace hotova. Vyzkoušejte ji a pokuste se zjistit jak pracuje.

11. Mimo seznamu řetězců uložených ve vlastnosti **Strings** může seznam obsahovat také seznam objektů, které jsou uloženy ve vlastnosti **Objects**. Vlastnost **Objects** je také indexovaná, je to indexovaný seznam objektů. Jestliže používáme řetězce v seznamu, není nutno mít také objekty; seznam nedělá nic s objekty, pokud k nim specificky nepřistupujeme. Builder pouze drží informaci o objektu a my s ní pracujeme. Některé seznamy řetězců přidané objekty ignorují (protože pro ně nemají význam). Např. seznam řetězců v komponentě **Memo** neukládá přidané objekty. Přestože vlastnosti **Objects** můžeme přiřadit libovolný typ objektu, často používáme bitové mapy. Důležité je zapamatovat si, že řetězec a objekt tvoří pár. Pro každý řetězec je přiřazen objekt; implicitně objekt je **NULL**. Seznam řetězců nevlastní přiřazené objekty, tzn. uvolnění objektu seznamu neuvolňuje objekty přiřazené řetězcům.

    Práci s objekty provádíme obdobně, jako práci s řetězci. Např. k jednotlivým objektům přistupujeme indexací vlastnosti **Objects** a to stejně jako u vlastnosti **Strings**. Pro přidání, vložení a lokalizaci objektu pou-

žíváme metody *AddObject*, *InsertObject* a *IndexOfObject*. Metody *Delete*, *Clear* a *Move* pracují s prvkem seznamu jako s celkem, tzn. zrušení prvku ruší jak řetězec, tak i odpovídající objekt. Metody *LoadFromFile* a *SaveToFile* operují pouze s řetězci, neboť pracují s textovými soubory. Pro přiřazení objektu k existujícímu řetězci přiřadíme objekt vlastnosti **Objects** se stejným indexem. Např. jestliže seznam řetězců jména *Ovoce* obsahuje řetězec „jablko“ a chceme mu přiřadit objekt bitové mapy nazvaný *JablkoBitmap* použijeme následující příkaz:

```
Ovoce->Objects[Ovoce->IndexOf("jablko ")] = JablkoBitmap;
```

Objekt můžeme také přidat současně s řetězcem. Např.

```
Ovoce->AddObject("jablko ", JablkoBitmap);
```

Nelze ale přidat objekt bez odpovídajícího řetězce.

Pokuste se vytvořit aplikaci, ve které budete zobrazovat obrázky bitových map. Zadáte jméno souboru bitové mapy, zobrazíte ji a bude moci zadat zobrazení další bitové mapy. Současně budete vytvářet seznam jmen souborů zobrazených bitových map a samotných objektů bitových map (můžeme jej považovat za seznam historie) a volbou v tomto seznamu opět zobrazte příslušnou bitovou mapu (bez opětovného načítání ze souboru).

12. Dále si ukážeme aplikaci zobrazující seznam písem dostupných v systému (jméno písma bude v seznamu zobrazeno pomocí tohoto písma; použijeme seznam s uživatelským zobrazováním prvků). Začneme novou aplikaci. Na formulář umístíme k hornímu okraji komponentu **Label** s textem *Písma systému:*, většinu plochy formuláře bude zabírat komponenta **ListBox** a ve spodní části formuláře bude další komponenta **Label** (se jménem vybraného písma). U komponenty **ListBox** nastavíme vlastnost **Style** na *lbOwnerDrawVariable* a vytvoříme obsluhy zobrazující seznam. Obsluha událost **OnMeasureItem** (událost zjišťující výšku prvku) bude vypadat takto:

```
void __fastcall TForm1::ListBox1MeasureItem(TWinControl *Control,
      int Index, int &Height)
{
  ListBox1->Canvas->Font->Name =ListBox1->Items->Strings[Index].c_str();
  ListBox1->Canvas->Font->Size = 0;
  Height = ListBox1->Canvas->TextHeight("Wg") +2;
}
```

Obsluha **OnDrawItem** (zobrazení prvku seznamu) vypadá takto:

```
void __fastcall TForm1::DrawItem(TWinControl *Control,
      int Index, TRect &Rect, TOwnerDrawState State)
{
  ListBox1->Canvas->FillRect(Rect);
  ListBox1->Canvas->Font->Name =ListBox1->Items->Strings[Index].c_str();
  ListBox1->Canvas->Font->Size = 0;
  ListBox1->Canvas->TextOut(Rect.Left+1, Rect.Top+1,
                            ListBox1->Items->Strings[Index].c_str());
}
```

Obsluha kliknutí na prvku seznamu pouze zobrazí jméno vybraného prvku ve spodní komponentě **Label**:

```
FontLabel->Caption =
                ListBox1->Items->Strings[ListBox1->ItemIndex].c_str();
```

Zbývá ještě vytvořit obsluhu **OnCreate** formuláře. Je tvořena příkazem:

```
ListBox1->Items = Screen->Fonts;
```

Tím je aplikace hotova. Můžeme ji vyzkoušet. Zjistěte jak pracují a k čemu slouží jednotlivé metody.

13. V této kapitole se ještě seznámíme s používáním další ze základních komponent a to s komponentou **ScrollBar**. Přímé použití této komponenty je vzácné. Typickým příkladem je umožnění toho, aby si uživatel mohl vybrat celočíselnou hodnotu z určitého rozsahu. Začneme s vývojem nové aplikace. Na formulář umístíme tři komponenty **ScrollBar** a vlevo od každé z nich komponentu **Label**. Každý posuvník se vztahuje k jedné ze tří základních barev a budeme s nimi určovat složky barvy formuláře. Posuvníky mají mnoho zvláštních vlastností. **Min** a **Max** používáme k určení rozsahu možných hodnot, **Position** obsahuje současnou pozici, vlastnost **LargeChange** a **SmallChange** určují krok, o který se změní hodnota. V našem příkladě budou všechny posuvníky mít možné hodnoty od 0 do 255, počáteční hodnotu 192 a krok bude 25 a 1. Obsluha události **OnScroll** posuvníku změní hodnotu **Caption** sousední komponenty **Label** a barvu formuláře. Např. pro první posuvník bude tvořena příkazy:

```
Label1->Caption := 'Červená: ' + IntToStr(ScrollPos);
Color := RGB(ScrollBar1->Position, ScrollBar2->Position, ScrollBar3->Position);
```

Další posuvníky budou pro složky zelená a modrá. Jejich obsluhy budou podobné. Vytvořte je sami. Funkce *RGB* vezme tři hodnoty složek a vytvoří hodnotu s kódem výsledné barvy. Povšimněte si, že obsluha udá-

losti **OnScroll** má tři parametry: *Sender*, *ScrollCode* (druh události) a *ScrollPos* (poslední pozici posuvníku). Druh události může být použit pro velice přesné ovládání akcí uživatele. Jeho hodnota indikuje, zda uživatel posunuje ukazatel (*scTrack*, *scPosition* a *scScroll*), zda kliknul na šipky nebo na lištu v jednom ze dvou možných směrů (*scLineUp*, *scLineDown*, *scPageUp* a *scPageDown*) a jestli se nepokusil o posun mimo rozsah (*scTop* a *scBottom*).

14. Vytvořte aplikaci, kde na formulář umístíte posuvník a komponentu **Label** s nějakým textem. Pomocí posuvníku měňte velikost písma komponenty **Label** (od 8 do 72).

7. Textový editor

1. V této kapitole se budeme zabývat vytvářením složitější aplikace. Bude to MDI aplikace textového editoru. Aplikaci budeme vytvářet v těchto krocích: Vytvoříme všechny služby MDI, v podřízeném okně vytvoříme textový editor, umožníme aplikaci zavádění a ukládání souborů, přidáme možnost tisku a zajistíme rozumné ukončení aplikace (dotaz na uložení neuloženého souboru při uzavírání aplikace).

   Vývoj naší aplikace začneme volbou **File | New Application**, a změníme tyto vlastnosti vytvořeného formuláře: **Name** na *FrameForm*, **Caption** na *Textový editor* a **FormStyle** na *fsMDIForm*. K vytvoření podřízeného formuláře přidáme nový prázdný formulář do projektu a u tohoto formuláře nastavíme vlastnosti: **Name** na *EditForm*, **Caption** na *Nepojmenovaný* a **FormStyle** na *fsMDIChild*. Soubory projektu je vhodné nyní uložit. Soubor formuláře rámu uložíme pod jménem MDIFrame.CPP, soubor podřízeného okna pod jménem MDIEdit.CPP a projektový soubor pod jménem TEXTEDIT.MAK (pro projekt vytvoříme speciální adresář).

   Ve formuláři rámu vytvoříme tuto nabídku (čísla v závorkách jsou hodnoty vlastnosti **GroupIndex**):

```
&Soubor (0)          &Okno (9)
&Nový                &Dlaždice
&Otevřít ...         &Kaskáda
-----------          &Uspořádat ikony
&Konec
```

   Dále vytvoříme nabídku podřízeného formuláře (pro prvky nabídky Úpravy zadáme také hodnoty vlastnosti **ShortCut**; jsou uvedeny ve třetím sloupci):

```
&Soubor (0)          Úpr&avy (1)                        &Znak (1)
&Nový                &Vyjmout          Ctrl+X          &Vlevo
&Otevřít ...         &Kopírovat        Ctrl+C          Vp&ravo
&Uložit              V&ložit           Ctrl+V          &Na střed
Uložit j&ako ...     Vy&mazat          Ctrl+D          ---------
&Zavřít              --------                          &Lámání řádků
-----------          Vybr&at vše       Ctrl+A          -------------
&Tisk ...                                              &Písmo ...
Nastavení tisku ...
---------------
&Konec
```

   Pro prvky **Vlevo** a **Lámání řádků** z nabídky **Znak** nastavíme vlastnost **Checked** na *True* (zajištění implicitního chování editoru).

   Naše aplikace provádí při otevření podřízeného okna náhradu nabídky **Soubor** rámu stejnojmennou nabídkou podřízeného okna. K sdílení obsluh událostí mezi těmito nabídkami, vytvoříme obsluhu události k prvku nabídky formuláře rámu a v obsluze události odpovídacího prvku nabídky podřízeného formuláře voláme tuto obsluhu události prvku formuláře rámu. V naší aplikaci vytvoříme obsluhu události **OnClick** prvku nabídky **Soubor | Konec** formuláře rámu:

```
Close();
```

   Tato obsluha v podřízeném formuláři bude tvořena příkazem:

```
FrameForm->Konec1Click(Sender);
```

   Nyní, když se pokusíme spustit vytvářenou aplikaci, překladač se zastaví na této nové obsluze a vypíše signalizaci oznamující, že *FrameForm* je nedefinovaný. Musíme tedy vložit hlavičkový soubor pro jednotku **FrameForm** do souboru *MDIEdit.CPP*. Provedeme to příkazem **File | Include Unit Hdr…**.

   Podřízený formulář musíme vyřadit se seznamu automatického vytváření formulářů (volba **Options | Project**). Podřízený formulář budeme vytvářet v obsluze události volby **Soubor | Nový**. V hlavním formuláři bude tato obsluha tvořena příkazem:

```
EditForm = new TEditForm(this);
```

   a v podřízeném formuláři:

```
FrameForm->Nov1Click(Sender);
```

Jestliže nyní překládáme naši aplikaci, pak Builder generuje chybové hlášení, neboť v jednotce hlavního formuláře nezná *TEditForm*. K vložení hlavičkového souboru jednotky podřízeného formuláře do hlavního formuláře použijeme opět volbu **File | Include Unit Hdr…**.

Podřízené okno budeme potřebovat také uzavřít. Vytvoříme tedy obsluhu volby **Soubor | Zavřít** (v podřízeném formuláři). Je tvořena příkazem:

```
Close();
```

a dále je nutno vytvořit obsluhu **OnClose** podřízeného formuláře s příkazem:

```
Action = caFree;
```

Vytvoříme také obsluhy voleb v nabídce **Okno**. Obsluha volby **Dlaždice** je tvořena příkazem:

```
Tile();
```

obsluha volby **Kaskada** tvoří příkaz:

```
Cascade();
```

a obsluhu **Uspořádej ikony** tvoří:

```
ArrangeIcons();
```

Vlastnost **WindowMenu** formuláře rámu nastavíme na *Okno1*. Tím dosáhneme přidávání seznamu otevřených oken do nabídky **Okno**. Titulek podřízeného okna by mohl obsahovat jméno otevřeného souboru. Pro uložení této informace přidáme do soukromé části deklarace **EditForm** (do souboru MdiEdit.h):

```
AnsiString PathName;
```

a na začátek jednotky podřízeného okna vložíme konstantu s implicitním jménem okna:

```
const AnsiString DefaultFileName = AnsiString("Nepojmenovaný");
```

Tuto konstantu použijeme v obsluze **OnCreate** podřízeného okna takto:

```
PathName = DefaultFileName;
```

Později, když do editoru zavedeme soubor, přiřadíme **PathName** aktuální jméno souboru. To indikuje, že editor obsahuje uložený soubor. Tím máme první krok vytváření aplikace hotov. Je zajištěno standardní MDI chování aplikace.

2. Nyní se budeme zabývat vytvářením textového editoru v podřízeném okně. Do podřízeného okna vložíme komponentu **RichEdit**. Tato komponenta se podobá komponentě **Memo**, ale nabízí větší možnosti formátování textu. U komponenty **RichEdit** nastavíme tyto vlastnosti: **Align** na *alClient*, **BorderStyle** na *bsNone*, **ScrollBars** na *ssVertical*, **Name** na *Editor* a vlastnost **Lines** vyprázdníme**.**

   V aplikaci textového editoru uživatel nastavuje zarovnávání textu pomocí voleb v nabídce **Znak**. Vytvoříme tedy obsluhu události **OnClick** pro prvek nabídky **Znak | Vlevo** a připojíme tuto obsluhu i k dalším dvěma prvkům této nabídky. Obsluha bude tvořena příkazy (pracujeme zde s vlastností **Paragraph** editoru; můžeme měnit zarovnávání i části textu):

```
Vlevo1->Checked = false;
Vpravo1->Checked = false;
Nasted1->Checked = false;
if (dynamic_cast<TMenuItem *>(Sender) !=0)
  dynamic_cast <TMenuItem *>(Sender) ->Checked = true;
if (Vlevo1->Checked) Editor->Paragraph->Alignment = taLeftJustify;
else if (Vpravo1->Checked) Editor->Paragraph->Alignment= taRightJustify;
else if (Nasted1->Checked) Editor->Paragraph->Alignment = taCenter;
```

   Komponenta editoru může obsahovat vodorovný, svislý nebo oba posuvníky. Při návrhu nastavíme počáteční hodnotu vlastnosti **ScrollBars** pro komponentu editoru na *ssVertical*. Když aplikaci spustíme, editor obsahuje pouze svislý posuvník (lámání řádků je povoleno a text není širší než aktuální šířka editoru). Zakážeme-li lámání řádků, pak potřebujeme i vodorovný posuvník. V obsluze volby **Znak | Lámání řádků** musíme tedy mimo nastavení vlastnosti **WordWrap** nastavovat i vlastnost **ScrollBars**. Provedeme to takto:

```
Editor->WordWrap = !Editor->WordWrap;
if (Editor->WordWrap) Editor->ScrollBars = ssVertical;
                  else Editor->ScrollBars = ssBoth;
Lmndk1->Checked = Editor->WordWrap;
```

   Textové editory umožňují přenášet vybraný text mezi dokumenty. Objekt **Clipboard** zaobaluje schránku Windows a poskytuje metody pro práci se schránkou. Objekt schránky je deklarován v jednotce *Clipbrd*. V naší aplikaci (přesněji řečeno v souboru **MDIEdit.h**) přidáme na začátek (za ostatní direktivy **include**) direktivu:

```
#include <vcl\Clipbrd.hpp>
```

   Dříve než můžeme vložit text do schránky, musíme jej vybrat. Zvýraznění vybraného textu je zabudováno v editační komponentě. Komponenta editoru má několik vlastností a metod pro práci s vybraným textem.

SelText je vlastnost použitelná pouze při běhu aplikace a obsahuje řetězec reprezentující text vybraný v komponentě. Metoda **SelectAll** vybírá všechen text komponenty. Metody **SelLength** a **SelStart** vracejí délku a počáteční pozici vybraného textu. Vytvoříme obsluhu volby **Úpravy | Vybrat vše** (jinak se výběrem textu nemusíme zabývat). Obsluhu tvoří příkaz:

```
Editor->SelectAll();
```

Obsluhy voleb práce se schránkou a volby **Úpravy | Vymazat** jsou vždy tvořeny jedním příkazem. Vytvořte tyto obsluhy sami (každou obsluhu tvoří jeden z následujících příkazů):

```
Editor->CutToClipboard();
Editor->CopyToClipboard();
Editor->PasteFromClipboard();
Editor->ClearSelection();
```

Nyní již naše aplikace může pracovat se schránkou. K editoru přidáme i místní nabídku s prvky **Vyjmout**, **Kopírovat** a **Vymazat** a připojíme k nim již existující obsluhy událostí. Bylo by ale vhodné, aby v případě, kdy nemáme vybrán žádný text, aby nebyly přístupné prvky nabídky **Vyjmout**, **Kopírovat** a **Vymazat** a v případě, kdy není nic uloženo ve schránce, aby byl nepřístupný prvek **Vložit**. Týká se to obou nabídek. Toto zakazování voleb v nabídce provedeme v obsluze události **OnClick** pro prvek nabídky **Úpravy** a použijeme ji i pro obsluhu **OnPopup** místní nabídky. Obsluha bude tvořena příkazy:

```
bool JeVybrano;
Vloit1->Enabled = Clipboard()->HasFormat(CF_TEXT);
Vloit2->Enabled = Vloit1->Enabled;
JeVybrano = Editor->SelLength > 0;
Vyjmout1->Enabled = JeVybrano;
Vyjmout2->Enabled = JeVybrano;
Koprovat1->Enabled = JeVybrano;
Koprovat2->Enabled = JeVybrano;
Vymazat1->Enabled = JeVybrano;
```

Metoda **HasFormat** vrací hodnotu určující, zda schránka obsahuje objekt, text nebo obrázek. Parametrem *CF_TEXT* určujeme, že nás zajímá pouze, zda ve schránce je text a podle vrácené hodnoty povolujeme nebo zakazujeme volbu **Vložit**. Tím jsme dokončili vytváření vlastního editoru.

3. V dalším kroku vývoje aplikace se budeme zabývat použitím různých dialogových oken Windows. Využijeme je při otevírání souborů, změně písma a ukládání souborů. Pro změnu písma využijeme komponentu **FontDialog** (umístíme ji na **EditForm**). Obsluha volby **Znak | Písmo** bude tvořena příkazy:

```
FontDialog1->Font= Editor->Font;
if (FontDialog1->Execute())
  Editor->SelAttributes->Assign(FontDialog1->Font);
```

Neměníme zde vlastnost **Font** ale **SelAttributes**. Tím dosáhneme změnu písma pouze vybraného textu. Dále se budeme zabývat otevíráním souborů. Využijeme komponentu **OpenDialog** (tentokrát ji musíme přidat na formulář rámu) a nazveme ji *OpenFileDialog*. Nastavíme u ní filtr souborů a to takto: *Textový soubor Rich (*.rft)*, *Textový soubor (*.txt)* a *Všechny soubory (*.*)*. Vytvoříme obsluhu volby **Soubor | Otevřít** pro formulář rámu:

```
if(OpenFileDialog->Execute()){
  EditForm = new TEditForm(this);
  EditForm->Open(OpenFileDialog->FileName);
}
```

a voláme tuto obsluhu v obsluze stejné volby podřízeného formuláře:

```
FrameForm->Otevt1Click(Sender);
```

V předchozí obsluze voláme metodu **Open**. Tuto metodu musíme také vytvořit. Do veřejné části deklarace **TEditForm** přidáme:

```
void __fastcall Open(const AnsiString AFileName);
```

a do jednotky podřízeného formuláře tuto metodu zapíšeme:

```
void __fastcall TEditForm::Open(const AnsiString AFileName)
{
  PathName = AFileName;
  Caption = ExtractFileName(AFileName);
  Editor->Lines->LoadFromFile(PathName);
  Editor->SelStart = 0;
  Editor->Modified = false;
}
```

Když editor zavede soubor, nastavíme hodnotu vlastnosti **SelStart** editoru na nulu. Toto určuje aktuální pozici kurzoru (a také začátek vybraného textu). Implicitně po zavedení souboru je tato vlastnost nastavena na

konec posledního řádku. Vlastnost **Modified** určuje, zda text souboru byl modifikován. Tuto vlastnost budeme testovat před uzavřením okna. Dále se budeme zabývat ukládáním souboru. Nejjednodušší případ je opětovné uložení existujícího souboru. Když uživatel zvolí **Soubor | Uložit**, aplikace zapíše soubor pod stejným jménem. Pouze pokud se takto snažíme uložit nepojmenovaný soubor, jsme dotázáni na jeho jméno. Obsluha volby **Soubor | Uložit** (v podřízeném formuláři) tedy bude tvořena příkazy:

```
if(Caption == DefaultFileName) UloitJako1Click(Sender);
else{
  Editor->Lines->SaveToFile(PathName);
  Editor->Modified = false;
}
```

Testujeme, zda soubor má přiřazené jméno a pokud nemá, pak voláme obsluhu volby **Uložit jako**. Pro zadávání jména ukládaného souboru přidáme na podřízený formulář komponentu **SaveDialog** a změníme její jméno na *SaveFileDialog*. Obsluha volby **Soubor | Uložit jako** bude tvořena příkazy:

```
SaveFileDialog->FileName = PathName;
if (SaveFileDialog->Execute() ){
  PathName= SaveFileDialog->FileName;
  Caption = ExtractFileName(PathName);
  Uloit1Click(Sender);
}
```

Tím jsme dokončili další etapu vytváření naší aplikace, tj. používání dialogových oken Windows.

4. V dalším kroku vývoje aplikace se budeme zabývat tiskem. Builder poskytuje objekt **Printer**, který zapouzdřuje většinu chování tiskárny a zjednodušuje tak spolupráci s tiskárnou. Pro použití tohoto objektu musíme přidat hlavičkový soubor *vcl\Printers.hpp* do hlavičkového souboru podřízeného formuláře. Tiskové funkce v naší aplikaci budeme používat pouze částečně. Umožníme tisknout pouze obsah celého souboru nebo vybraný text. Pro nastavení tiskárny vložíme komponentu **PrinterSetupDialog** na podřízený formulář. Tato komponenta pracuje přímo s konfiguračními soubory Windows a my nemusíme dělat prakticky nic. Obsluha volby **Soubor | Nastavení tisku** bude tvořena příkazem:

   ```
   PrinterSetupDialog1->Execute();
   ```

   Na podřízený formulář dále vložíme komponentu **PrintDialog** a vytvoříme obsluhu volby **Soubor | Tisk** s těmito příkazy:

   ```
   if (PrintDialog1->Execute()){
     try {
       Editor->Print(PathName);
     }
     catch(...){
       Printer()->EndDoc();
       throw;
     }
   }
   ```

   Nyní naše aplikace má schopnost tisknout textové soubory.

5. V posledním kroku vývoje naší aplikace se budeme zabývat uložením zatím neuložených modifikovaných souborů při ukončení aplikace. Pro zabránění uzavření formuláře vytvoříme obsluhu události **OnCloseQuery** formuláře. Před uzavřením formuláře metoda **Close** generuje událost **OnCloseQuery**. Tato událost má parametr **CanClose**, který určuje zda formulář může být uzavřen. Vytvoříme tedy obsluhu události **OnCloseQuery** s těmito příkazy:

   ```
   char buffer[255];
   if (Editor->Modified) {
     Set<TMsgDlgBtn,0,8> temp_set;
     temp_set<< mbYes<<mbNo <<mbCancel;
     sprintf(buffer,"Uložit změny do %s?",PathName.c_str());
     switch(MessageDlg(buffer, mtConfirmation,temp_set,0)) {
       case mrYes: Uloit1Click(this);
       case mrCancel: CanClose=false;
     }
   }
   ```

   V obsluze není nutno testovat stisknutí tlačítka **No**, protože implicitní hodnota **CanClose** je *True.* Aplikace textového editoru je nyní hotova.

6. Pokuste se nějakým způsoben vylepšit naši aplikaci textového editoru. Např. přidejte k aplikaci paletu nástrojů, kterou budete moci zadávat zarovnávání textu a další běžné činnosti.

7. V další aplikaci se více seznámíme s komponentou **RichText**. Vytvoříme opět textový editor, tentokrát se nebude jednat o MDI aplikaci. Začneme s vývojem nové aplikace a pro formulář nastavíme tyto vlastnosti: **Caption** na *Demonstrace ovladače RichText* a formulář zvětšíme (**Width** nastavíme na 606 a **Height** na 407). Formuláři můžeme také přiřadit nějakou ikonu. Pro aplikaci vytvoříme následující nabídku:

```
&Soubor              Úpr&avy              &Nápověda
&Nový                &Zpět                &Obsah
&Otevřít ...         ----------           &Hledání prvku nápovědy ...
&Uložit              &Vyjmout             &Jak používat nápovědu
Uložit j&ako ...     &Kopírovat           ----------------------
---------------      V&ložit              O &aplikaci
&Tisk ...            ----------
---------------      &Písmo ...
&Konec
```

Do vlastností **Hint** jednotlivých prvků nabídky vložíme příslušné texty nápověd. Např. pro prvek **Nový** to bude text *Vytvoř nový soubor*, pro **Otevřít** *Otevři existující soubor* a dále použijeme texty: *Ulož aktuální soubor*, *Ulož aktuální soubor pod novým jménem*, *Vytiskni aktuální soubor*, *Ukonči tuto aplikaci*, *Zruš poslední akci*, *Vystřihni výběr do schránky*, *Kopíruj výběr do schránky***,** *Vlož obsah schránky*, *Zvol písmo*, *Zobraz nápovědu*, *Hledej nápovědný soubor pro prvek*, *Jak používat nápovědu* a *Informace o aplikaci*. Formulář vybavíme také paletou nástrojů. Pro oddělení nabídky od palety nástrojů vložíme na formulář komponentu **Bevel** a nastavíme u ní tyto vlastnosti: **Height** na 2, **Align** na *alTop* a **Shape** na *bsTopLine*. Na formulář vložíme komponentu **Panel** a nastavíme její vlastnosti takto: **Caption** vyprázdníme, **Height** nastavíme na 32, **Align** na *alTop*, **BevelOuter** na *bvNone*, **ParentShowHint** na *False* a **ShowHint** na *True*. Na panel vložíme ovladače podle následujícího obrázku:

Pro jednotlivé ovladače zde nastavíme vlastnosti podle následující tabulky (všechny mají hodnotu vlastnosti **Top** nastavenou na 5):

Name	**Left**	**Hint**	**AllowAllUp**	**GroupIndex**	**Tag**
OpenButton	8	Otevři			
SaveButton	33	Ulož			
PrintButton	58	Tiskni			
UndoButton	89	Zruš změnu			
CutButton	114	Vyjmi			
CopyButton	139	Kopíruj			
PasteButton	164	Vlož			
BoldButton	380	Tučně	True	1	
ItalicButton	406	Kurzíva	True	2	
UnderlineButton	432	Podtrženo	True	3	
LeftButton	464	Zarovnání vlevo	True	4	
CenterButton	490	Centrovat	True	4	2
RightButton	516	Zarovnání vpravo	True	4	1
BulletsButton	547	Odrážky	True	5	

Na panel vložíme také komponentu **ComboBox** u které nastavíme vlastnosti: **Left** na 194, **Width** na 131, **Text** vyprázdníme, **Height** nastavíme na 21 a **ItemHeight** na 13. Panel obsahuje dále komponentu **TEdit** s vlastnostmi: **Left** rovno 328, **Width** 31, **Height** 21 a **Text** 0. K tomuto ovladači připojíme komponentu **UpDown**, kde nastavíme: **Left** na 359, **Width** na 15, **Height** na 21, **Associate** na připojený editační ovladač, **Min** na 0, **Position** na 0 a **Wrap** na *False*. Tím je paleta nástrojů hotova.

Dále se pokusíme vytvořit pravítko. Bude tvořeno komponentou panel. Na formulář tedy vložíme další komponentu **Panel** a nastavíme u ní tyto vlastnosti: **Name** na *Ruler*, **Height** na 26, **Align** na *alTop*, **Alignment** na *taLeftJustify*, **BevelInner** na *bvLowered*, **BevelOuter** na *bvNone*, **BorderWidth** na 1, písmo nastavíme na *Arial* a vlastnost **Caption** vyprázdníme. Na pravítko vložíme komponentu **Bevel** a nastavíme u ní: **Name** na *RulerLine*, **Left** na 4, **Top** na 12, **Width** na 579, **Height** na 2 a **Shape** na *bsTopLine*. Pravítko bude také obsahovat indikátory levého okraje, levého okraje prvního řádku a indikátor pravého okraje. Každý z těchto indikátorů bude tvořen komponentou **Label**. Jejich vlastnosti nastavíme podle následující tabulky (pro všechny nastavíme **AutoSize** na *False*, **DragCursor** na *crArrow* a **Font** na *Wingdings*):

Name	Left	Top	Caption
FirstInd	2	2	ę (bude zobrazeno ↓)
LeftInd	2	12	é (bude zobrazeno ↑)
RightInd	575	14	ń (bude zobrazeno ⇧)

Na formulář vložíme dále komponentu **StatusBar** a její vlastnost **SimplePanel** nastavíme na *True*. Vlastní editor bude tvořen komponentou **RichEdit**. Zde nastavíme **Align** na *alClient* a **ScrollBars** na *ssBoth*.

Dále na formulář vložíme **OpenDialog** (kde **Filter** nastavíme na *Textové soubory Rich (*.RTF)* a *Textové soubory (*.TXT)*, **SaveDialog** (stejné nastavení **Filter** jako u **OpenDialog**), **PrintDialog** a **FontDialog**. Tím máme všechny potřebné komponenty umístěny na formuláři.

8. Dále k aplikaci přidáme další formulář a vytvoříme z něj okno **O aplikaci**. Okno zmenšíme asi na poloviční rozměry a nastavíme pro něj tyto vlastnosti: **BorderStyle** na *bsDialog*, **Caption** na *O aplikaci editoru Rich* a **Position** na *poScreenCenter*. Formulář dále rozdělíme komponentou **Bevel** na dvě části. Bude opět ve tvaru vodorovné čáry tentokrát asi uprostřed formuláře. Do horní části vložíme ikonu programu a texty informující o jménu programu a jeho autorovi. Do spodní části umístíme pod sebe dvě komponenty **Label** s texty *Paměť dostupná z Windows* a *Použitá paměť*. Vpravo od těchto komponent umístíme další dvě komponenty **Label** pro zobrazování zjištěných informací. Jejich vlastnosti **Caption** nastavíme na *0* a **Name** horní komponenty změníme na *PhysMem* a spodní komponenty na *FreeRes*. Na formuláři bude ještě tlačítko, kde změníme **Caption** na *OK*, **Cancel** a **Default** na *True* a **ModalResult** na *mrCancel*. Tím je návrh tohoto formuláře hotov.

   Tento formulář bude obsahovat pouze obsluhu události **OnCreate**. Obsluha bude tvořena následujícími příkazy (zjistíme zde velikost dostupné paměti a procento její využití):

```
TMemoryStatus MS;
MS.dwLength = sizeof(MS);
GlobalMemoryStatus(&MS);
PhysMem->Caption = FormatFloat((AnsiString)"#,###' KB'",
                   MS.dwTotalPhys / 1024);
LPSTR lpMemLoad = new char[5];
sprintf(lpMemLoad,"%d %%",MS.dwMemoryLoad);
FreeRes->Caption = (AnsiString)lpMemLoad;
delete [] lpMemLoad;
```

   Tento formulář dále vyřadíme ze seznamu automaticky vytvářených formulářů.

9. V dalším kroku vývoje naší aplikace se budeme zabývat vytvářením obsluh pro volby v nabídce hlavního formuláře. Začneme nejprve jednoduchými obsluhami. Obsluha volby **Úpravy | Vyjmout** bude tvořena příkazem:

```
RichEdit1->CutToClipboard();
```

   Stejnou obsluhu přiřadíme i stisku tlačítka **CutButton**. Pro **Úpravy | Kopírovat** to bude příkaz (i pro stisk tlačítka **CopyButton**):

```
RichEdit1->CopyToClipboard();
```

   Obdobně pro **Úpravy | Vložit** (a tlačítko **PasteButton**)

```
RichEdit1->PasteFromClipboard();
```

   Pro volbu **Úpravy | Písmo** (tlačítko pro tuto funkci nemáme) vytvoříme obsluhu s příkazy:

```
FontDialog1->Font->Assign(RichEdit1->SelAttributes);
if (FontDialog1->Execute()) CurrText()->Assign(FontDialog1->Font);
RichEdit1->SetFocus();
```

   Volba **Nápověda | O aplikaci** bude tvořena příkazy (do *Unit1* musíme vložit hlavičkový soubor *Unit2*):

```
Form2 = new TForm2(Application);
Form2->ShowModal();
delete Form2;
```

   Pro vytvoření dalších obsluh voleb v nabídce budeme potřebovat vytvořit další položky a metody formuláře. Do soukromé části deklarace formuláře vložíme tyto deklarace:

```
AnsiString FFileName;
bool FUpdating;
int FDragOfs;
bool FDragging;
TTextAttributes *__fastcall CurrText(void);
void __fastcall GetFontNames(void);
void __fastcall SetFileName(const AnsiString FileName);
void __fastcall CheckFileSave(void);
```

```
void __fastcall SetupRuler(void);
void __fastcall SetEditRect(void);
```
Na začátku programové jednotky hlavního formuláře uvedeme deklaraci následujících konstant:
```
const float RulerAdj = 4.0/3.0;
const int SireOkraje = 6;
```
Implementace uvedených metod bude vypadat takto (je zde i další pomocná funkce):
```
TTextAttributes *__fastcall TForm1::CurrText(void)
{
  return RichEdit1->SelAttributes;
}
int __stdcall EnumFontsProc(TLogFontA &LogFont,
                            TTextMetricA &/*TextMetric*/,
                            int /*FontType*/, Pointer Data)
{
  ((TStrings *)Data)->Add((AnsiString)LogFont.lfFaceName);
  return 1;
}
void __fastcall TForm1::GetFontNames(void)
{
  HDC hDC = GetDC(0);
  void * cTmp = (void *) ComboBox1->Items;
  EnumFonts(hDC, NULL, (FONTENUMPROC) EnumFontsProc, (long) cTmp );
  ReleaseDC(0,hDC);
  ComboBox1->Sorted = True;
}
void __fastcall TForm1::SetFileName(const AnsiString FileName)
{
  LPSTR lpBuf = new char[MAX_PATH];
  sprintf(lpBuf, "%s - %s", ExtractFileName(FileName).c_str(),
          Application->Title.c_str());
  Caption = (AnsiString)lpBuf;
  FFileName = FileName;
  delete lpBuf;
}
void __fastcall TForm1::CheckFileSave(void)
{
  if (RichEdit1->Modified) {
    switch(MessageBox(Handle, "Uložit změny?","Potvrzení",
           MB_YESNOCANCEL | MB_ICONQUESTION)) {
      case ID_YES    : Uloit1Click(this);
      case ID_CANCEL : Abort();
    };
  }
}
void __fastcall TForm1::SetupRuler(void)
{
  int iCtr = 1;
  char sTmp[201];
  while (iCtr < 200) {
    sTmp[iCtr] = 9;
    iCtr++;
    sTmp[iCtr] = '|';
    iCtr++;
  }
  Ruler->Caption = (AnsiString)sTmp;
}
void __fastcall TForm1::SetEditRect(void)
{
  TRect Rct = Rect(SireOkraje, 0, RichEdit1->ClientWidth-SireOkraje,
                   ClientHeight);
  SendMessage(RichEdit1->Handle, EM_SETRECT, 0, long(&Rct));
}
```
Nyní již můžeme uvést obsluhy dalších voleb v nabídce. Obsluha volby **Soubor | Nový** bude tvořena příkazy:

```
CheckFileSave();
SetFileName((AnsiString)"Nepojmenovaný");
RichEdit1->Lines->Clear();
RichEdit1->Modified = False;
```

Obsluhu volby **Soubor | Otevřít** a tlačítka **OpenButton** tvoří:

```
CheckFileSave();
if (OpenDialog1->Execute()) {
  RichEdit1->Lines->LoadFromFile(OpenDialog1->FileName);
  SetFileName(OpenDialog1->FileName);
  RichEdit1->SetFocus();
  RichEdit1->Modified = False;
  RichEdit1->ReadOnly = OpenDialog1->Options.Contains(ofReadOnly);
}
```

Obdobně pro **Soubor | Uložit** a tlačítko **SaveButton** to budou příkazy:

```
if (!strcmp(FFileName.c_str(), "Nepojmenovaný"))
  Uloitjako1Click(Sender);
else {
  RichEdit1->Lines->SaveToFile(FFileName);
  RichEdit1->Modified = False;
}
```

Obsluhu **Soubor | Uložit jako** tvoří příkazy:

```
if (SaveDialog1->Execute()) {
  RichEdit1->Lines->SaveToFile(SaveDialog1->FileName);
  SetFileName(SaveDialog1->FileName);
  RichEdit1->Modified = False;
}
```

Obsluhu volby **Soubor | Tisk** a tlačítka **PrintButton** tvoří příkaz:

```
if (PrintDialog1->Execute() ) RichEdit1->Print( FFileName );
```

a obsluhu **Soubor | Konec** příkaz:

```
Close();
```

Pokuste se pochopit jak tyto obsluhy a další metody pracují. Další obsluhy voleb nám již srozumitelné asi nebudou. Obsluha volby **Úpravy | Zpět** a stisknutí tlačítka **UndoButton** je tvořena:

```
if (RichEdit1->HandleAllocated())
  SendMessage(RichEdit1->Handle, EM_UNDO, 0, 0);
```

obsluhu **Nápověda | Obsah** tvoří příkaz:

```
Application->HelpCommand(HELP_CONTENTS, 0);
```

Obsluhu **Nápověda | Hledání prvku nápovědy** tvoří:

```
Application->HelpCommand(HELP_PARTIALKEY, (long) "");
```

a obsluha **Nápověda | Jak používat nápovědu** je tvořena:

```
Application->HelpCommand(HELP_HELPONHELP, 0);
```

Tím jsme dokončili vytváření obsluh voleb v nabídce. Zbývá nám vyřešit ještě stisknutí několika tlačítek na paletě. Obsluha stisku tlačítka **BoldButton** obsahuje příkazy:

```
if (!FUpdating) {
  if (BoldButton->Down)
    CurrText()->Style = CurrText()->Style << fsBold;
  else
    CurrText()->Style = CurrText()->Style >> fsBold;
}
```

Pro tlačítko **ItalicButton** to bude:

```
if (!FUpdating) {
  if (ItalicButton->Down)
    CurrText()->Style = CurrText()->Style << fsItalic;
  else
    CurrText()->Style = CurrText()->Style >> fsItalic;
}
```

a pro tlačítko **UnderlineButton**:

```
if (!FUpdating) {
  if (UnderlineButton->Down)
    CurrText()->Style = CurrText()->Style << fsUnderline;
  else
    CurrText()->Style = CurrText()->Style >> fsUnderline;
```

```
}
```
Pro všechna tři tlačítka zarovnávání bude použita obsluha s příkazy:
```
if (!FUpdating) {
  TControl *oAliBtn = (TControl*)(Sender);
  RichEdit1->Paragraph->Alignment = (TAlignment)oAliBtn->Tag;
}
```
Obsluha stisku posledního tlačítka, tj. tlačítka **BulletsButton** je tvořena příkazy:
```
if (!FUpdating)
  RichEdit1->Paragraph->Numbering = (TNumberingStyle)BulletsButton->Down;
```
Zbývají ještě obsluhy **OnChange** editačního ovladače velikosti písma a kombinovaného ovladače jména písma. Obsluha editačního ovladače je tvořena příkazy:
```
int fontsize = atoi(Edit1->Text.c_str());
if ((!FUpdating) && (fontsize)) {
  if (fontsize < 1) {
    ShowMessage("The number must be between 1 and 1638.");
    Edit1->Text = 1;
  }
  else if (fontsize > 1638) {
    ShowMessage("The number must be between 1 and 1638.");
    Edit1->Text = 1638;
  }
  CurrText()->Size = atoi(Edit1->Text.c_str());
}
```
a obsluha kombinovaného okna obsahuje:
```
if (!FUpdating) {
  CurrText()->Name = ComboBox1->Items->Strings[ComboBox1->ItemIndex];
}
```
Dále se budeme zabývat vytvářením obsluh dalších událostí. V události **OnCreate** formuláře budeme také měnit obsluhu události **OnHint** aplikace. Do soukromé části deklarace třídy formuláře tedy vložíme
```
void __fastcall ShowHint(TObject *Sender);
```
a do jednotky formuláře zapíšeme změněnou obsluhu.
```
void __fastcall TForm1::ShowHint(TObject* /*Sender*/)
{
  StatusBar1->SimpleText = Application->Hint;
}
```
Nyní již můžeme vytvořit obsluhu události **OnCreate** formuláře. Bude tvořena těmito příkazy:
```
Application->OnHint = &ShowHint;
OpenDialog1->InitialDir = ExtractFilePath(ParamStr(0));
SaveDialog1->InitialDir = OpenDialog1->InitialDir;
GetFontNames();
SetupRuler();
SelectionChange(this);
```
Pro komponentu **RichEdit** vytvoříme obsluhu události **OnSelectionChange** (nazveme ji *SelectionChange*) s těmito příkazy:
```
char sizebuf[6];
try {
  FUpdating = True;
  FirstInd->Left = int(RichEdit1->Paragraph->FirstIndent*RulerAdj)-
                       4+SireOkraje;
  LeftInd->Left = int((RichEdit1->Paragraph->LeftIndent+
              RichEdit1->Paragraph->FirstIndent)*RulerAdj)-4+SireOkraje;
  RightInd->Left = Ruler->ClientWidth-6-
          int((RichEdit1->Paragraph->RightIndent+SireOkraje)*RulerAdj);
  BoldButton->Down = RichEdit1->SelAttributes->Style.Contains(fsBold);
  ItalicButton->Down=RichEdit1->SelAttributes->Style.Contains(fsItalic);
  UnderlineButton->Down=RichEdit1->SelAttributes->Style.Contains(fsUnderline);
  BulletsButton->Down = bool(RichEdit1->Paragraph->Numbering);
  Edit1->Text = itoa(RichEdit1->SelAttributes->Size, sizebuf, 10);
  ComboBox1->Text = RichEdit1->SelAttributes->Name;
  switch((int)RichEdit1->Paragraph->Alignment) {
    case 0: LeftButton->Down   = True; break;
    case 1: RightButton->Down  = True; break;
```

```
    case 2: CenterButtonAlign->Down = True; break;
  }
}
catch (...) {
  FUpdating = False;
}
FUpdating = False;
```

Obsluha **OnResize** pravítka bude tvořena příkazem:

```
RulerLine->Width = (int)Ruler->ClientWidth - (RulerLine->Left*2);
```

a obsluhu **OnResize** formuláře tvoří příkazy:

```
SetEditRect();
SelectionChange(Sender);
```

Pro formulář ještě vytvoříme obsluhu události **OnPaint** s příkazem

```
SetEditRect();
```

a obsluhu události **OnCloseQuery** tvořenou příkazy:

```
try {
  CheckFileSave();
}
catch (...) {
  CanClose = False;
}
```

Tím máme většinu aplikace hotovou. Zbývá dokončit ještě ovládání pravítka. to dokončíme v dalším bodě. Pokuste se pochopit, co a jak provádějí jednotlivé obsluhy.

10. V tomto bodě se budeme zabývat ovládáním indikátorů okrajů na pravítku. Pro všechny tři indikátory okrajů na pravítku vytvoříme obsluhy událostí **OnDouseDown**, **OnMouseMove** a **OnMouseUp**. První dvě z těchto obsluh jsou pro všechny tři indikátory společné. Obsluhu **OnMouseDown** tvoří příkazy:

```
TLabel * oTmpLabel = (TLabel *)Sender;
FDragOfs = oTmpLabel->Width / 2;
oTmpLabel->Left = oTmpLabel->Left+X-FDragOfs;
FDragging = True;
```

a obsluhu **OnMouseMove** příkazy:

```
if (FDragging) {
  TLabel * oTmpLabel = (TLabel *)Sender;
  oTmpLabel->Left = oTmpLabel->Left+X-FDragOfs;
}
```

Obsluhu **OnMouseUp** indikátoru prvního řádku tvoří příkazy:

```
FDragging = False;
RichEdit1->Paragraph->FirstIndent = int((FirstInd->Left+
                                    FDragOfs-SireOkraje) / RulerAdj);
LeftIndMouseUp(Sender, Button, Shift, X, Y);
```

pro indikátor levého okraje použijeme příkazy:

```
FDragging = False;
RichEdit1->Paragraph->LeftIndent = int((LeftInd->Left+
      FDragOfs-SireOkraje)/RulerAdj)-RichEdit1->Paragraph->FirstIndent;
SelectionChange(Sender);
```

a pro indikátor pravého okraje příkazy:

```
FDragging = False;
RichEdit1->Paragraph->RightIndent = int((Ruler->ClientWidth-
           RightInd->Left+FDragOfs-2) / RulerAdj)-2*SireOkraje;
SelectionChange(Sender);
```

Nyní je již naše aplikace hotova. Můžeme ji vyzkoušet.

11. Naše předchozí aplikace obsahovala v nabídce volby pro používání nápovědy, ale žádnou nápovědu se nám nepodařilo získat. Nemáme zatím vytvořen soubor nápovědy. Vytváření nápovědy zahájíme vytvořením souboru s jednotlivými prvky nápovědy. Tento soubor musí být ve formátu RichText (musíme také použít příponu RTF). Otevřeme tedy nový soubor v některém textovém editoru, který umožňuje tento formát používat (např. MS Word; je vhodné si nejprve vyzkoušet vytvoření malého souboru nápovědy) a zapíšeme jednotlivé prvky nápovědy. Každý prvek nápovědy ukončíme tvrdým koncem stránky. Na začátek každého prvky připojíme potřebné poznámky pod čarou (podle následujícího popisu). Identifikaci prvku nápovědy zadáváme jako poznámku pod čarou označenou znakem #. Jméno identifikace má některá omezení: může obsahovat mezery, ale nesmí začínat nebo končit mezerou, nesmí být delší než 255 znaků a nesmí obsahovat

znaky # = + @ * % !. Jestliže používáme mapování, pak nesmí začínat číslicí. Prvek nápovědy může také obsahovat titulek. Titulek zadáváme jako poznámku pod čarou označenou znakem $. Titulek může mít nejvýše 255 znaků.

K označení prvku jako cíl ALink (vyhledávání prvků obsahujících A-klíčová slova) umístíme na začátek prvku poznámku pod čarou označenou A. Jako poznámku pod čarou v tomto případě zapíšeme jedno nebo více klíčových slov, které oddělujeme středníkem. Klíčové slovo nesmí obsahovat více než 255 znaků, nesmí obsahovat znak odřádkování a mezery před a za klíčovým slovem jsou odstraněny. K vytvoření indexové položky pro prvek nápovědy umístíme na začátek prvku nápovědy poznámku pod čarou označenou K a do poznámky zapíšeme seznam klíčových slov (mají stejná omezení jako u poznámky pod čarou A). V případě indexů lze vytvořit i druhou úroveň indexů. Provedeme to tak, že v poznámce zapíšeme klíčové slovo první úrovně indexu, které ukončíme středníkem a bezprostředně za tímto středníkem zapíšeme opět klíčové slovo první úrovně, následované čárkou nebo dvojtečkou a po mezeře zapíšeme klíčové slovo druhé úrovně indexu ukončené středníkem (např. `makro; makro, knihovna;`).

Prvek nápovědy můžeme také zařadit do vyhledávání. Provedeme to vložením poznámky pod čarou označené +, na začátek prvku nápovědy. Jako poznámku pak zapíšeme vyhledávací kód, který nesmí obsahovat znaky # = + @ * % ! a nesmí být delší než 50 znaků.

K vytvoření skoku na prvek nápovědy nebo vysvětlení hesla umístíme kurzor přímo za text nebo bitovou mapu, pro kterou chceme skok provést a zapíšeme identifikaci prvku na který chceme skočit. Vybereme textový prvek, pro který chceme provést skok a aplikujeme na něj styl dvojitého podtržení (pro skok na další prvek nápovědy) nebo styl jednoduchého podržení (pro vysvětlení hesla). Dále musíme vybrat následující identifikaci prvku a aplikujeme na něj styl **Skryté**.

Pro přidání bitové mapy k prvku nápovědy vložíme bitovou mapu přímo do souboru na místo jejího zobrazení. Jestliže chceme použít vícenásobné instance bitových map, vytvoříme spojení k jejich souboru pomocí následující syntaxe:

```
{bmx filename.bmp}
```

kde x specifikuje způsob zarovnávání (c - bitová mapa se bere jako znak, l - zarovnání k levému okraji a r - zarovnávání k pravému okraji).

Tímto způsobem vytvoříme textový soubor typu Rich s prvky nápověd. Ve výše uvedeném popisu tohoto souboru jsou uvedeny pouze základní možnosti vytváření nápovědy. Jestliže Vás zajímá kompletní popis, podívejte se do nápovědy Help Workshopu. Vytvořený soubor s prvky nápověd musíme zpracovat Help Workshopem a vytvořit soubor nápovědy projektu.

Spustíme Help Workshop (jedná se o soubor **hcw.exe**; je umístěn v adresáři *CBuilder\Help\Tools*) a zvolíme **File | New**. V zobrazeném okně zvolíme *Help Project*, zapíšeme jméno projektového souboru nápovědy (přípona HPJ) a stiskneme OK. Tím vytvoříme základ projektového souboru nápovědy. K tomuto souboru musíme dále přidat vytvořené soubory s prvky nápověd (soubory typu Rich), specifikujeme adresáře s použitými soubory bitových map a můžeme zadat ještě mnoho dalších věcí. Po dokončení definice projektového souboru nápovědy stiskneme **Save and Compile**, čímž projektový soubor uložíme a přeložíme. Tím je vytváření souboru nápovědy ukončeno.

Pro vytvoření nápovědy pro naši aplikaci textového editoru Rich použijeme soubory s prvky nápověd **Richedit.rtf**, **Keys.rtf** a **Term.rtf**. Tyto soubory jsou obsažené na přiložené disketě (všechny soubory, které se týkají vytvářené nápovědy překopírujeme do adresáře projektu naší aplikace). Prohlédněte si je, podívejte se také na použité poznámky pod čarou. V tomto případě použijeme již vytvořené soubory, při vytváření nápovědy v dalších aplikacích je již budeme vytvářet sami. Spustíme Help Workshop, zvolíme **File | New**, vybereme volbu *Help Project* a zadáme jméno projektového souboru (v našem případě zvolíme **RichEdit**, umístíme jej do adresáře, ve kterém máme projekt naší aplikace). Do projektového souboru vložíme výše uvedené tři soubory typu Rich. Provedeme to stiskem tlačítka **Files** a po stisku **Add** tyto soubory specifikujeme. Naše nápověda používá také několik souborů bitových map (jsou také na přiložené disketě). Pokud tyty soubory máme ve stejném adresáři, jako je umístěn projektový soubor nápovědy, pak nemusíme zadávat adresář bitových map. Mohli bychom zadat ještě některé další volby (např. použití komprese u souboru nápovědy, určit tlačítka použitá v okně nápovědy), ale zatím se spokojíme pouze s tímto zadáním. Vytváření souboru nápovědy ukončíme stiskem tlačítka **Save and Compile**.

Soubor nápovědy nyní již máme vytvořen. Potřebujeme jej ale ještě připojit k naši aplikaci. Provedeme to vložením jména nápovědného souboru do vlastnosti **HelpFile** objektu aplikace. Vytvoříme např. obsluhu události **OnActivate** hlavního formuláře s příkazy:

```
Application->HelpFile = "RICHEDIT.HLP";
RichEdit1->SetFocus();
```

Nyní je již v naší aplikaci možno používat nápovědu. Podíváme-li se na příkazy, které nápovědu zobrazují, pak vidíme použití metody **HelpCommand** objektu aplikace. Co z nápovědy je zobrazeno určuje první pa-

rametr této metody. Při použití parametru HELP_CONTENTS je zobrazen prvek nápovědy s obsahem nápovědy (druhý parametr metody v tomto případě nemá význam). Parametr HELP_HELPONHELP určuje, že bude zobrazen obsah souboru nápovědy **WinHlp32** (soubor popisující jak používat nápovědu). Druhý parametr metody v tomto případě opět nemá význam. Parametr HELP_PARTIALKEY určuje hledání prvku nápovědy podle indexů. Druhý parametr metody v tomto případě je adresa implicitního hledaného řetězce.

Tím jsme vývoj této aplikace dokončily. Vyzkoušejte používání nápovědy.

12. Další aplikace je velmi jednoduchá. Ukážeme si v ní, jaké standardní kurzory můžeme používat. Na formulář umístíme pouze komponentu **Label** s textem *Zvol kurzor* a komponentu kombinovaného ovladače. U kombinovaného ovladače nastavíme tyto vlastnosti: **Text** na *crDefault* a do vlastnosti **Items** vložíme 21 řetězců s názvy jednotlivých kurzorů (z následující deklarace). Vytvoříme ještě obsluhu události **OnChange** kombinovaného ovladače. Bude tvořena příkazem:

```
static int Kurzory[] = {
        crDefault, crNone, crArrow, crCross, crIBeam,
        crSize, crSizeNESW, crSizeNS, crSizeNWSE,
        crSizeWE, crUpArrow, crHourGlass, crDrag,
        crNoDrop, crHSplit, crVSplit, crMultiDrag,
        crSQLWait, crNo, crAppStart, crHelp};
Cursor = TCursor(Kurzory[ComboBox1->ItemIndex]);
```

Aplikace je hotova a můžeme si prohlédnout tvary jednotlivých kurzorů. V kombinovaném ovladači vybereme název kurzoru a při přesunu ukazatele myši nad prázdnou část formuláře se zvolený kurzor zobrazí.

13. Dále se pokusíme vytvořit aplikaci, která nám umožní měřit počet zapsaných znaků (za minutu) do editačního okna. V této aplikaci se seznámíme s další komponentou. Začneme s vývojem nové aplikace. Formulář zvětšíme na rozměry asi 480 x 350 a nastavíme u něj vlastnosti **BorderStyle** na *bsSingle*, **BorderIcons** na *[biSystemMenu, biMinimize]* a **Caption** na *Počet znaků za minutu*. Do horního levého rohu formuláře vložíme komponentu **Label** s textem *Pro zahájení testu začněte psát.* a zvětšíme její písmo na 10. Ke spodnímu okraji formuláře umístíme komponentu **ProgressBar** (umístíme ji asi 5 mm od okrajů formuláře) a nastavíme její vlastnosti takto: **Min** na 0, **Max** na 60 a **Step** na 1. Nad tuto komponentu umístíme další **Label**, jeho text vyprázdníme a písmo zvětšíme na 14. Vedle horní komponenty **Label** umístíme ještě dvě tlačítka (s texty *Vyprázdnit* a *Konec*). Prostředek formuláře vyplníme komponentou **RichEdit**, kterou vybavíme svislým posuvníkem (komponentu vyprázdníme). Vlastnost **ActiveControl** formuláře nastavíme na **EditRich**. Na formulář umístíme ještě komponentu **Timer** a nastavíme u ní vlastnost **Enabled** na *False*. Obsluha stisku tlačítka **Konec** je tvořena příkazem:

```
Application->Terminate();
```

a obsluhu stisku tlačítka **Vyprázdnit** tvoří příkazy:

```
Timer1->Enabled = False;
Label1->Caption = "Pro zahájení testu začněte psát.";
Label1->Font->Color = clBlack;
Label2->Caption = "";
RichEdit1->SetTextBuf("");
RichEdit1->ReadOnly = False;
ProgressBar1->Position = 0;
```

Pro komponentu **RichEdit** vytvoříme obsluhu události **OnKeyDown** (spustí měření času testu). Tato obsluha je tvořena příkazy:

```
if (RichEdit1->ReadOnly == false) {
  Timer1->Enabled = True;
  Label1->Caption = "Test spuštěn";
  Label1->Font->Color = clGreen;
}
```

Dále vytvoříme pomocnou funkci, která nám spočítá počet zapsaných znaků (vytvoříme z ní soukromou metodu formuláře):

```
int __fastcall TForm1::PocetZnaku(void) {
  int I, Pocet=0;
  for (I = 0; I < RichEdit1->Lines->Count; I++)
    Pocet += RichEdit1->Lines[I]->Length();
  return Pocet;
}
```

Zbývá ještě vytvořit obsluhu **OnTimer** časovače. Tuto obsluhu tvoří příkazy:

```
if (ProgressBar1->Position == 59) {
  Timer1->Enabled = False;
  Label1->Caption = "Test dokončen";
```

```
  Label1->Font->Color = clRed;
  RichEdit1->ReadOnly = True;
  Label2->Caption = IntToStr(PocetZnaku()) + " znaků za minutu";
}
ProgressBar1->StepIt();
```

Aplikace je hotova. Pokuste se zjistit jak pracuje. Zápis prvního znaku do **RichEdit** spustí měření času. Vyzkoušejte.

14. Na závěr této kapitoly si ukážeme ještě některé možnosti používání mřížky. Velikost každého sloupce nebo řádku můžeme nastavit nezávisle na ostatních, protože vlastnosti **RowSize**, **ColWidth** a **RowHeight** jsou pole. Uživateli můžeme také umožnit změnu velikosti řádků a sloupců (volby *goColSizing* a *goRowSizing*), přetahování celých sloupců a řádků na nové pozice (*goColMoving* a *goRowMoving*), volbu automatické editace a rozsáhlejší výběry. Existuje také mnoho událostí vztahující se k mřížkám. Nejdůležitější je pravděpodobně **OnDrawCell**. V obsluze této události musíme nakreslit určité políčko mřížky (pokud se obsah mřížky nezobrazuje automaticky).

    V další aplikaci použijeme komponentu **StringGrid** (je to jediná komponenta vložená na formulář). Nastavíme zde vlastnosti **Align** na *alClient*, **DefaultDrawing** na *False* (umožní uživatelské zobrazování obsahu buněk) a **Options** na *[goFixedVertLine, goFixedHorzLine, goVertLine, goHorzLine, goDrawFocusSelected, goColSizing, goColMoving, goEditing]*. V naší mřížce budeme zobrazovat písma systému v různých velikostech. V obsluze **OnCreate** formuláře spočítáme počet sloupců a řádků a jejich velikosti. Tato obsluha tedy bude obsahovat příkazy:

```
int I, J;
StringGrid1->ColCount = Screen->Fonts->Count + 1;
StringGrid1->ColWidths[0] = 50;
for (I = 1; I <= Screen->Fonts->Count; I++) {
  StringGrid1->Cells[I] [0] = Screen->Fonts->Strings[I-1];
  StringGrid1->Canvas->Font->Name = StringGrid1->Cells[I] [0];
  StringGrid1->Canvas->Font->Size = 32;
  StringGrid1->ColWidths[I]=StringGrid1->Canvas->TextWidth("AaBbYyZz");
};
StringGrid1->RowCount = 26;
for (I = 1; I <= 25; I++) {
  StringGrid1->Cells [0] [I] = IntToStr(I+7);
  StringGrid1->RowHeights[I] = 15 + I*2;
  for (J = 1; J <= StringGrid1->ColCount; J++)
    StringGrid1->Cells [J] [I] = "AaBbYyZz";
}
StringGrid1->RowHeights[0] = 25;
```

Pro vlastní zobrazování obsahu mřížky musíme dále vytvořit obsluhu **OnDrawCell** mřížky. Tato obsluha obsahuje příkazy:

```
if (Col == 0) StringGrid1->Canvas->Font->Name = "Arial";
else StringGrid1->Canvas->Font->Name = StringGrid1->Cells[Col] [0];
if (Row == 0) StringGrid1->Canvas->Font->Size = 14;
else StringGrid1->Canvas->Font->Size = Row + 7;
if (State.Contains(gdSelected)) StringGrid1->Canvas->Brush->Color=clHighlight;
else if (State.Contains(gdFixed)) StringGrid1->Canvas->Brush->Color=clBtnFace;
     else StringGrid1->Canvas->Brush->Color = clWindow;
StringGrid1->Canvas->TextRect(Rect,Rect.Left,Rect.Top,
                              StringGrid1->Cells[Col] [Row]);
if (State.Contains(gdFocused)) StringGrid1->Canvas->DrawFocusRect(Rect);
```

Tím je naše aplikace hotova. Pokuste se zjistit význam jednotlivých příkazů v obou obsluhách událostí. Aplikaci vyzkoušejte. Je vhodné formulář zvětšit.

7. Kreslení při běhu aplikace

1. Jsou tři možnosti získání grafického obrázku v aplikaci Builderu: můžeme vložit předem vytvořený obrázek během návrhu, vytvořit jej použitím grafického ovladače při návrhu nebo jej dynamicky kreslit při běhu aplikace. V této kapitole se budeme zabývat kreslením při běhu aplikace a to buď do okna nebo do nějaké komponenty. Když kreslíme v aplikaci Builderu, můžeme kreslit na plátno (canvas) objektu; je to vhodnější než kreslit přímo na objekt. Plátno je vlastnost objektu a je to samotný objekt se svými vlastními vlastnostmi. Objekt plátna má čtyři důležité vlastnosti: pero pro kreslení čar, štětec pro vyplňování tvarů, písmo pro zápis textů a pole bodů reprezentujících obraz. Vlastnost **Font** se používá běžným způsobem. Zde se budeme zabývat ostatními třemi vlastnostmi. Plátna jsou přístupná pouze při běhu aplikace a všechnu práci s nimi je nutno provádět pomocí kódu.

   Každé plátno má indexovanou vlastnost nazvanou **Pixels**, která reprezentuje jednotlivé barevné body obrázku na plátně. K vlastnosti **Pixels** přistupujeme přímo pouze když potřebujeme zjistit nebo nastavit barvu konkrétního bodu. **Pixels** představuje kreslící plochu plátna a pero nebo štětec pracují se skupinou bodů.

   Body plátna si můžeme představit jako dvourozměrné pole, kde každý prvek pole má určitou barvu. Můžeme číst nebo nastavovat barvu jednotlivých bodů. Pro získání barvy jistého bodu, přistoupíme k vlastnosti **Pixels** plátna použitím indexů x- a y-ové souřadnice tohoto bodu. Počátkem souřadnicového systému je horní levý roh plátna. Např. následující obsluha události reaguje na kliknutí na značce změnou barvy textu značky na barvu bodu o souřadnicích (10, 10) na formuláři:

```
void __fastcall TForm1::CheckBox1Click(TObject *Sender)
{
    CheckBox1->Font->Color = Canvas->Pixels[10][10];
}
```

   K nastavení barvy bodu přiřadíme hodnotu barvy vlastnosti **Pixels**, použitím x- a y-ové souřadnice bodu. Např. následující obsluha události reaguje na stisknutí tlačítka změnou barvy náhodného bodu formuláře na červenou:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Canvas->Pixels[random(ClientWidth)] [random(ClientHeight)] = clRed;
}
```

   Vlastnost **Pen** plátna řídí způsob zobrazování čar, včetně kreslení čar obrysů tvarů. Kreslení rovné čáry změní skupinu bodů, které jsou přímo mezi dvěma body. Samotné pero má čtyři vlastnosti, které můžeme měnit: **Color**, **Width**, **Style** a **Mode**. Hodnoty těchto vlastností určují jak pero změní body na čáře. Implicitně každé pero začíná s černou barvou, se šířkou 1 bod, plnou čarou a režimem nazvaným *pmCopy*, který přepisuje body na plátně. Barvu pera můžeme nastavit při běhu aplikace změnou vlastnosti **Color**. **Width** je celé číslo určující šířku čáry v bodech. **Style** umožňuje používat plnou čáru, tečkovanou čáru, čárkovanou čáru apod. **Mode** slouží k určení způsobu kombinace barvy pera s původní barvou v poli bodů.

   Aktuální kreslící pozice (pozice, kde jsme skončili s kreslením předchozí čáry) se nazývá pozice pera. Plátno ukládá svoji pozici pera do vlastnosti nazvané **PenPos**. Pozici pera ovlivňuje pouze kreslení čar; pro tvary a texty zadáváme všechny požadované souřadnice. K nastavení pozice pera voláme metodu **MoveTo** plátna. Např. k přesunu pozice pera do horního levého rohu plátna použijeme následující kód:

```
Canvas->MoveTo(0, 0);
```

   Kreslení čáry metodou **LineTo** také přesouvá pozici pera do koncového bodu čáry.

   Vlastnost **Brush** plátna řídí způsob vyplňování oblastí, včetně vyplňování tvarů. Vyplňování oblasti štětcem způsobí změnu většího počtu určených bodů. Štětec má tři vlastnosti, které můžeme měnit: **Color**, **Style** a **Bitmap**. Hodnoty těchto vlastností určují způsob vyplňování tvarů nebo jiných oblastí plátna. Implicitně každý štětec začíná bílou barvou, plným stylem a bez vzoru bitové mapy. Barvu štětce můžeme nastavit při běhu aplikace pomocí vlastnosti **Color**. **Style** slouží k určení způsobu kombinace barvy štětce s původní barvou plátna. **Bitmap** slouží ke specifikaci obrázku pro štětec k použití jako vzor pro vyplňování tvarů a jiných oblastí.

   Na plátno můžeme kreslit dva typy čar: přímé čáry a lomené čáry. Přímá čára je čára spojující dva body. Lomená čára je řetězec přímých čar, které na sebe navazují. Všechny čary kreslíme pomocí pera. Ke kreslení přímých čar na platně používáme metodu **LineTo** plátna. **LineTo** kreslí čáru z aktuální pozice pera do bodu určujícího koncový bod čáry. Plátno kreslí čaru pomocí svého pera. Např. následující metoda nakreslí úhlopříčky formuláře při malování formuláře (uvedené příkazy tvoří obsluhu **OnPaint** formuláře):

```
Canvas->MoveTo(0, 0);
Canvas->LineTo(ClientWidth, ClientHeight);
```

```
Canvas->MoveTo(0, ClientHeight);
Canvas->LineTo(ClientWidth, 0);
```

Událost **OnPaint** vznikne, když Windows zjistí, že je potřeba formulář překreslit. Mimo samostatných čar, plátno může také kreslit lomené čáry, což je skupina několika propojených čárových segmentů. Pro kreslení lomené čáry na plátno, voláme metodu **PolyLine** plátna. Parametr předávaný metodě **PolyLine** je pole bodů. Tato metoda provede **MoveTo** do prvního bodu a **LineTo** do každého následujícího bodu. Následující příkaz např. kreslí na formuláři kosočtverec:

```
Canvas->Polyline(OPENARRAY(POINT,(Point(0,0), Point(50,0), Point(75,50),
                                  Point(25,50), Point(0,0))));
```

V tomto příkladu je použit otevřený parametr typu pole. Jako parametr můžeme předat libovolné pole bodů, ale snadnější je vytvořit pole pomocí makra OPENARRAY.

Plátna mají metody pro kreslení čtyř typů tvarů. Plátno kreslí obrys tvaru svým perem a vnitřek tvaru vyplňuje svým štětcem. Pro kreslení obdélníku nebo elipsy na plátnu, voláme metodu **Rectangle** nebo **Ellipse** plátna a předáme souřadnice ohraničujícího obdélníku. Metoda **Rectangle** kreslí ohraničující obdélník, **Ellipse** kreslí elipsu vloženou do obdélníku. Následující příkazy např. kreslí obdélník do horní levé čtvrtiny formuláře a pak do stejné oblasti nakreslí elipsu:

```
Canvas->Rectangle(0, 0, ClientWidth / 2, ClientHeight / 2);
Canvas->Ellipse(0, 0, ClientWidth / 2, ClientHeight / 2);
```

Pro kreslení obdélníků se zaoblenými rohy na plátno, voláme metodu **RoundRect** plátna. První čtyři parametry předané **RoundRect** určují ohraničující obdélník (stejně jako u **Rectangle**) a další dva parametry určují zaoblení rohů. Následující příkaz např. kreslí obdélník se zaoblenými rohy do horní levé čtvrtiny formuláře a zaoblené rohy jsou tvořeny čtvrtinou kružnice o poloměru 10 bodů:

```
Canvas->RoundRect(0,0,ClientWidth / 2,ClientHeight / 2, 10, 10);
```

Pro kreslení mnohoúhelníku na plátno, voláme metodu **Polygon** plátna. Mnohoúhelník přebírá pole bodů jako svůj jediný parametr a tyto body propojuje čarou nakreslenou perem (spojí také poslední bod s prvním k uzavření mnohoúhelníku). Po nakreslení čar je použit štětec k vyplnění mnohoúhelníku. Následuje příklad použití (pravoúhlý trojúhelník):

```
Canvas->Polygon(OPENARRAY(POINT, (Point(0, 0), Point(0, ClientHeight),
                  Point(ClientWidth, ClientHeight))));
```

Mimo kreslení čar a tvarů, které jsou zde popsány, plátno také poskytuje některé specializované grafické možnosti, jako kruhový oblouk, kruhová výseč apod. Můžeme také vyplňovat oblasti a kopírovat obrázky na jiné plátno. Informace o těchto možnostech najdeme v nápovědě.

Vyzkoušejte použití výše uvedených příkazů.

2. Dále se pokusíme vytvořit aplikaci grafického editoru. Jedná se opět o složitější aplikaci a budeme ji vytvářet v několika krocích. Nejprve se seznámíme s používáním událostí myši pro kreslení.

   Začneme s vývojem nové aplikace. Vytvoříme obsluhu události **OnMouseDown** formuláře k nastavení aktuální kreslící pozice na souřadnice, kde uživatel stiskl tlačítko myši:

   ```
   Canvas->MoveTo(X, Y);
   ```

   Stisknutí tlačítka myši nyní nastaví pozici pera, tj. nastaví počáteční bod čáry. Ke kreslení čáry je ještě zapotřebí bod, kde uživatel tlačítko myši uvolní. To určíme v obsluze události **OnMouseUp**. Tato událost je obvykle zaslána objektu, nad kterým bylo tlačítko myši stisknuto. V naši aplikaci vytvoříme obsluhu události **OnMouseUp** formuláře a předané souřadnice použijeme jako koncový bod čáry. Tato obsluha události bude vypadat takto:

   ```
   Canvas->LineTo(X, Y);
   ```

   Aplikaci můžeme vyzkoušet a nakreslit několik čar. Událost **OnMouseMove** se při pohybu myši vyskytuje periodicky. Událost je zasílána objektu, nad kterým bylo tlačítko myši stisknuto. V našem příkladu použijeme událost přemístění myši pro kreslení čáry podle pohybu myši. To dosáhneme použitím následující obsluhy události **OnMouseMove** formuláře:

   ```
   Canvas->LineTo(X, Y);
   ```

   Nyní jestliže spustíme aplikaci, pak pohybem myši po formuláři kreslíme čáru. Musíme ještě zajistit, aby kreslení probíhalo pouze při stisknutém tlačítku. V našem příkladu, je zapotřebí, aby formulář udržoval informaci o tom, zda tlačítko myši je stisknuto. K tomuto účelu do deklarace typu formuláře přidáme do veřejné části položku **Kreslit** typu bool a nastavíme její hodnotu při stisku tlačítka myši. Později budeme potřebovat další dvě položky, obě typu *TPoint*. Jedná se o položky **Pocatek** a **Presun**.

   ```
   bool Kreslit;
   TPoint Pocatek, Presun;
   ```

V položce **Kreslit** budeme udržovat informaci o tom, zda kreslit. Při stisku tlačítka ji nastavíme na *True* a při jeho uvolnění na *False*. V tomto smyslu změníme již vytvořené obsluhy události. Pro **OnMouseDown** použijeme příkazy:

```
Kreslit = True;
Canvas->MoveTo(X, Y);
```

a pro **OnMouseUp** příkazy:

```
Canvas->LineTo(X, Y);
Kreslit = False;
```

Dále musíme modifikovat obsluhu události **OnMouseMove** tak, aby kreslení probíhalo pouze tehdy, když **Kreslit** má hodnotu *True*:

```
if (Kreslit) Canvas->LineTo(X, Y);
```

Nyní kreslení probíhá pouze mezi stiskem tlačítka a jeho uvolněním. Pokaždé, při přesunu myši, obsluha události **OnMouseMove** volá **LineTo**, které mění pozici pera. Při uvolnění tlačítka myši již tedy neznáme počáteční bod čáry. Abychom po skončení kreslení mohli propojit počáteční bod křivky s koncovým a křivku tak uzavřít, použijeme položku **Pocatek** k uložení počátečního bodu křivky. Změníme tedy obsluhy událostí takto. Pro **OnMouseDown** použijeme:

```
Kreslit = True;
Canvas->MoveTo(X, Y);
Pocatek = Point(X, Y);
```

a pro **OnMouseUp**:

```
Canvas->MoveTo(Pocatek.X, Pocatek.Y);
Canvas->LineTo(X, Y);
Kreslit = False;
```

Obsluha události **OnMouseMove** je nyní napsána tak, že jsou kresleny čáry z poslední pozice myši do aktuální pozice. Můžeme to změnit tak, aby kreslené čáry začínaly v bodě stisku tlačítka myši:

```
if (Kreslit){
  Canvas->MoveTo(Pocatek.X, Pocatek.Y);
  Canvas->LineTo(X, Y);
}
```

Vyzkoušejte. Pro rozumnou práci této aplikace bychom potřebovali, aby vždy před nakreslením další čáry byla předchozí čára smazána. Využijeme k tomu položku **Presun**, kde bude uložen koncový bod předchozí nakreslené čáry. Naše obsluhy událostí změníme takto. Pro **OnMouseDown** použijeme:

```
Kreslit = True;
Canvas->MoveTo(X, Y);
Pocatek = Point(X, Y);
Presun = Point(X, Y);
```

a pro **OnMouseMove**:

```
if (Kreslit) {
  Canvas->Pen->Mode = pmNotXor;
  Canvas->MoveTo(Pocatek.X, Pocatek.Y);
  Canvas->LineTo(Presun.X, Presun.Y);
  Canvas->MoveTo(Pocatek.X, Pocatek.Y);
  Canvas->LineTo(X, Y);
}
Presun = Point(X,Y);
Canvas->Pen->Mode = pmCopy;
```

Nyní po spuštění aplikace se námi kreslená čára průběžně zobrazuje a zůstane zobrazená pouze ta, při které jsme uvolnili tlačítko myši. Je to dosaženo tím, že při zápisovém režimu *pmNotXor* druhé zobrazení stejné čáry tuto čáru smaže. Na závěr kreslení je opět nastaven implicitní zápisový režim *pmCopy*.

3. V dalším kroku vývoje naší aplikace se pokusíme vytvořit paletu nástrojů. Na formulář přidáme komponentu **Panel**, nastavíme její vlastnost **Align** na *alTop* (tím zabírá celou šířku formuláře), zrušíme její titulek a přidáme na ní tlačítka podle následujícího obrázku:

Tlačítka přejmenujeme takto (bráno po řadě zleva): **LineButton**, **RectangleButton**, **EllipseButton**, **RoundRectButton**, **PenButton** a **BrushButton**. Přiřadíme jim také vhodné obrázky. U prvního tlačítka nastavíme jeho vlastnost **Down** na *True* a z prvních čtyřech urychlovacích tlačítek vytvoříme skupinu (nastavíme jejich **GroupIndex** na 1). Tlačítka Pera a Štětce v našem příkladu (poslední dvě tlačítka) jsou jedno

tlačítkové skupiny, které pracují jako přepínací. Nastavíme jejich vlastnosti **AllowAllUp** na *True* a vlastnost **GroupIndex** pro **PenButton** na 2 (pro **BrushButton** na 3).

Budeme se snažit vytvořenou paletu nástrojů zapojit do práce. Zapamatujeme si zvolený nástroj a při další práci jej budeme používat. K zapamatování zvoleného nástroje je nejvhodnější použít výčtový typ. Přidáme tedy před deklaraci typu formuláře deklaraci výčtového typu **TKresliciNastroj** a položku **KresliciNastroj** tohoto typu vložíme do veřejné části deklarace typu formuláře:

```
enum TKresliciNastroj {knLine, knRectangle, knEllipse, knRoundRect};

TKresliciNastroj KresliciNastroj;
```

Změnu nástroje provádíme v reakci na událost **OnClick** stisknutého urychlovacího tlačítka. Vytvoříme tedy následující obsluhy událostí:

```
void __fastcall TForm1::LineButtonClick(TObject *Sender)
{
  KresliciNastroj = knLine;
}
//---------------------------------------------------------------------------
void __fastcall TForm1::RectangleButtonClick(TObject *Sender)
{
  KresliciNastroj = knRectangle;
}
//---------------------------------------------------------------------------
void __fastcall TForm1::EllipseButtonClick(TObject *Sender)
{
  KresliciNastroj = knEllipse;
}
//---------------------------------------------------------------------------
void __fastcall TForm1::RoundRectButtonClick(TObject *Sender)
{
  KresliciNastroj = knRoundRect;
}
```

Nyní již můžeme říci formuláři, který typ nástroje má použít a je nutno ještě formulář naučit jak jej má použít. Kreslící nástroj se používá v obsluhách událostí **OnMouseUp** a **OnMouseMove**. Změníme tyto obsluhy aby využívali zvolený nástroj (jeden ze čtyř možných). Pro větvení budeme používat příkaz **switch**.

```
void __fastcall TForm1::FormMouseUp(TObject *Sender, TMouseButton Button,
                                    TShiftState Shift, int X, int Y)
{
  switch (KresliciNastroj){
    case knLine: {
              Canvas->MoveTo(Pocatek.X, Pocatek.Y);
              Canvas->LineTo(X, Y);
              break;
            }
    case knRectangle: {
              Canvas->Rectangle(Pocatek.X, Pocatek.Y, X, Y);
              break;
            }
    case knEllipse: {
              Canvas->Ellipse(Pocatek.X, Pocatek.Y, X, Y);
              break;
            }
    case knRoundRect: {
              Canvas->RoundRect(Pocatek.X, Pocatek.Y, X, Y,
                                (Pocatek.X - X) /2, (Pocatek.Y - Y) / 2);
              break;
  }
  Kreslit = False;
}
void __fastcall TForm1::FormMouseMove(TObject *Sender, TShiftState Shift,
                                      int X, int Y)
{
  if (Kreslit) {
    Canvas->Pen->Mode = pmNotXor;
    switch (KresliciNastroj) {
```

```
      case knLine: {
                Canvas->MoveTo(Pocatek.X, Pocatek.Y);
                Canvas->LineTo(Presun.X, Presun.Y);
                Canvas->MoveTo(Pocatek.X, Pocatek.Y);
                Canvas->LineTo(X, Y);
                break;
              }
      case knRectangle: {
          Canvas->Rectangle(Pocatek.X, Pocatek.Y, Presun.X, Presun.Y);
          Canvas->Rectangle(Pocatek.X, Pocatek.Y, X, Y);
          break;
        }
      case knEllipse: {
          Canvas->Ellipse(Pocatek.X, Pocatek.Y, Presun.X, Presun.Y);
          Canvas->Ellipse(Pocatek.X, Pocatek.Y, X, Y);
          break;
        }
      case knRoundRect: {
          Canvas->RoundRect(Pocatek.X, Pocatek.Y, Presun.X, Presun.Y,
           (Pocatek.X - Presun.X) / 2, (Pocatek.Y - Presun.Y) / 2);
          Canvas->RoundRect(Pocatek.X, Pocatek.Y, X, Y,
           (Pocatek.X - X) / 2, (Pocatek.Y - Y) / 2);
          break;
        }
    }
    Presun = Point(X,Y);
  }
  Canvas->Pen->Mode = pmCopy;
}
```

V těchto metodách se několikrát opakuje stejný kód. Tento opakující se kód vložíme do samostatné metody, kterou v obou obsluhách použijeme. Přidáme deklaraci metody **KresliTvar** do veřejné části (na její konec) deklarace typu formuláře (mohli bychom ji vložit i do soukromé části):

```
void __fastcall KresliTvar(TPoint LevyHorni, TPoint PravySpodni,
                           TPenMode Rezim);
```

Definici této metody zapíšeme do programové jednotky. Metoda bude mít tento tvar:

```
void __fastcall TForm1::KresliTvar(TPoint LevyHorni, TPoint PravySpodni,
                                   TPenMode Rezim){
  Image->Canvas->Pen->Mode = Rezim;
  switch (KresliciNastroj){
    case knLine : {
      Canvas->MoveTo(LevyHorni.x, LevyHorni.y);
      Canvas->LineTo(PravySpodni.x, PravySpodni.y);
      break;
    }
    case knRectangle : {
      Canvas->Rectangle(LevyHorni.x, LevyHorni.y,
                        PravySpodni.x, PravySpodni.y);
      break;
    }
    case knEllipse : {
      Canvas->Ellipse(LevyHorni.x, LevyHorni.y,
                      PravySpodni.x, PravySpodni.y);
      break;
    }
    case knRoundRect : {
      Canvas->RoundRect(LevyHorni.x, LevyHorni.y,
        PravySpodni.x, PravySpodni.y,
        (LevyHorni.x -PravySpodni.x)/2, (LevyHorni.y - PravySpodni.y)/2);
      break;
    }
  }
}
```

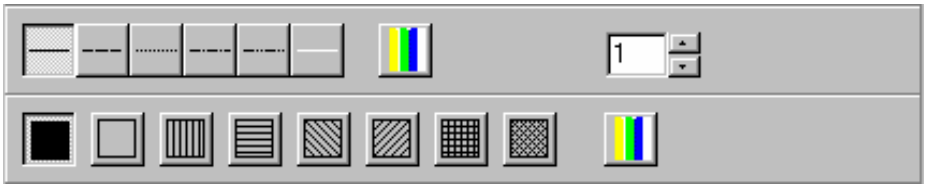
Tuto metodu nyní použijeme k dále uvedenému zjednodušení našich obsluh událostí:

```
void __fastcall TForm1::FormMouseUp(TObject *Sender, TMouseButton Button,
```

```
                                        TShiftState Shift, int X, int Y)
{
  if (Kreslit){
    KresliTvar (Pocatek, Point(X, Y), pmCopy);
    Kreslit = False;
  }
}
void __fastcall TForm1::FormMouseMove(TObject *Sender, TShiftState Shift,
                                      int X, int Y)
{
  if (Kreslit){
    KresliTvar(Pocatek, Presun, pmNotXor);
    Presun = Point(X, Y);
    KresliTvar (Pocatek, Presun, pmNotXor);
  }
}
```

Tím máme volbu kreslícího nástroje vyřešenou.

4. Grafický program, který kreslí pouze černobílé obrázky není moc zajímavý. V dalším kroku k naši aplikaci přidáme další panely nástrojů, jeden pro pero a druhý pro štětec. Paleta nástrojů nemusí být stále viditelná. Pokud aplikace obsahuje mnoho palet nástrojů, pak zobrazujeme pouze tu, kterou právě používáme. K vytvoření skryté palety nástrojů, přidáme komponentu **Panel** na formulář (nastavíme její vlastnost **Align** na *alTop*) a vlastnost panelu **Visible** nastavíme na *False*. V naši aplikaci přidáme na formulář dvě komponenty panelu (budou reprezentovat skryté palety), jeden nazveme **PenBar** a druhý **BrushBar** (zrušíme hodnoty jejich vlastností **Caption**). Výše uvedeným postupem nastavíme jejich vlastnosti. Obě vytvářené palety jsou zobrazeny na následujícím obrázku (nahoře je **PenBar** a dole **BrushBar**). Vlastnosti komponent nastavíme podle dalšího popisu (ovladače jsou uváděny zleva doprava). U prvních osmi tlačítek z **BrushBar** (vzory výplně) nastavíme vlastnosti **GroupIndex** na 1 a u prvních šesti tlačítek z **PenBar** (typy čar) nastavíme **GroupIndex** také na 1.



Tlačítka na **BrushBar** nazveme takto: **SolidBrush**, **ClearBrush**, **VerticalBrush**, **HorizontalBrush**, **FDiagonalBrush**, **BDiagonalBrush**, **CrossBrush**, **DiagCrossBrush** a **BrushColor**. U prvního tlačítka nastavíme vlastnost **Down** na *True*. Tlačítkům přiřadíme vhodné bitové mapy. Obdobně pro **PenBar** změníme jména tlačítek na: **SolidPen**, **DashPen**, **DotPen**, **DashDotPen**, **DashDotDotPen**, **ClearPen** a **PenColor**. První tlačítko opět stiskneme. Na této paletě je umístěn ještě editační ovladač, který nazveme **PenSize** a vlastnost **Text** u něj nastavíme na 1. K tomuto editačnímu ovladači připojíme komponentu **UpDown** (nastavíme její vlastnost **Associate** na *PenSize*).

K ukrytí nebo zobrazení palety nástrojů měníme její vlastnost **Visible** na *False* nebo *True*. Obvykle to provádíme v reakci na jistou událost nebo změnu režimu aplikace. V našem případě skrytí a zobrazování palety pera a štětce budeme ovládat tlačítky **PenButton** a **BrushButton**. Vytvoříme tedy následující obsluhy událostí:

```
void __fastcall TForm1::PenButtonClick(TObject *Sender)
{
  PenBar->Visible = PenButton->Down;
}
void __fastcall TForm1::BrushButtonClick(TObject *Sender)
{
  BrushBar->Visible = BrushButton->Down;
}
```

Jestliže nyní spustíme naši aplikaci, můžeme zobrazovat a ukrývat paletu pera a paletu štětce (jsou umístěny v horní části formuláře pod paletou kreslících nástrojů).

Styl pera určuje typ zobrazované čáry (plná, čárkovaná, tečkovaná atd.). Některé video ovladače nepodporují jiné styly pro čáry širší než 1 bod. Tyto ovladače všechny širší čáry kreslí plnou čarou (bez ohledu na zadaný styl). Ke změně stylu pera, nastavíme vlastnost pera **Style** na hodnotu určující požadovaný styl. Je možno použít některou z následujících konstant: *psSolid*, *psDash*, *psDot*, *psDashDot*, *psDashDotDot* nebo *psClear*. Můžeme tedy pro každé tlačítko určující styl vytvořit obsluhu události **OnClick** a přiřadit v ní odpovídající

konstantu vlastnosti **Style**. V našem programu budeme ale postupovat jiným způsobem. Pro všechna tato tlačítka vytvoříme sdílenou obsluhu události a stisknuté tlačítko v ní určíme pomocí parametru *Sender*. Vybereme tedy všech šest tlačítek stylu pera, na stránce událostí v Inspektoru objektů vybereme **OnClick**, do sloupce obsluhy zapíšeme *SetPenStyle* a stiskneme Enter. Builder generuje obsluhu události a přiřadí ji všem vybraným tlačítkům. Vygenerovanou obsluhu události pak doplníme takto:

```
if (Sender == SolidPen){ Canvas->Pen->Style = psSolid; }
else if (Sender == DashPen){ Canvas->Pen->Style = psDash; }
else if (Sender == DotPen){ Canvas->Pen->Style = psDot; }
else if (Sender == DashDotPen){ Canvas->Pen->Style = psDashDot; }
else if (Sender == DashDotDotPen){ Canvas->Pen->Style = psDashDotDot; }
else if (Sender == ClearPen){ Canvas->Pen->Style = psClear; }
```

Barva pera určuje barvu kreslené čáry, včetně obrysů kreslených tvarů. Pro změnu barvy pera změníme vlastnost pera **Color**. Barvu zadáváme volbou v dialogovém okně barev Windows. Na formulář tedy musíme umístit komponentu **ColorDialog**, ponecháme její implicitní jméno *ColorDialog1* a vytvoříme následující obsluhu události **OnClick** pro tlačítko **PenColor**:

```
ColorDialog1->Color = Canvas->Pen->Color;
if (ColorDialog1->Execute()){
  Canvas->Pen->Color = ColorDialog1->Color;
}
```

Šířka pera určuje tloušťku čáry v bodech. Ke změně šířky pera přiřadíme číselnou hodnotu vlastnosti pera **Width**. Vytvoříme obsluhu události **OnChange** editačního ovladače **PenSize** a to takto:

```
Canvas->Pen->Width = PenWidth->Position;
```

Styl štětce určuje výplňový vzor při vyplňování tvarů. Předdefinované styly zahrnují plnou plochu, prázdný styl a různé styly šrafování. Pro změnu stylu štětce, nastavíme jeho vlastnost **Style** na jednu z předdefinovaných hodnot: *bsSolid*, *bsClear*, *bsHorizontal*, *bsVertical*, *bsFDiagonal*, *bsBDiagonal*, *bsCross* nebo *bsDiagCross*. V naši aplikaci opět vytvoříme sdílenou obsluhu události pro všech osm tlačítek stylu štětce (obdobně jako pro styl pera):

```
if (Sender == SolidBrush){ Canvas->Brush->Style = bsSolid; }
else if (Sender == ClearBrush){ Canvas->Brush->Style = bsClear; }
else if (Sender == HorizontalBrush){Canvas->Brush->Style = bsHorizontal;}
else if (Sender == VerticalBrush){Canvas->Brush->Style = bsVertical;}
else if (Sender == FDiagonalBrush){ Canvas->Brush->Style = bsFDiagonal; }
else if (Sender == BDiagonalBrush){ Canvas->Brush->Style = bsBDiagonal; }
else if (Sender == CrossBrush){ Canvas->Brush->Style = bsCross; }
else if (Sender == DiagCrossBrush){ Canvas->Brush->Style = bsDiagCross; }
```

Barva štětce určuje barvu výplně. Při její změně měníme hodnotu vlastnosti štětce **Color**. V naši aplikaci změnu barvy štětce vyřešíme obdobně jako změnu barvy pera. Obsluha stisku tlačítka **ColorBrush** bude tvořena příkazy:

```
ColorDialog1->Color = Canvas->Brush->Color;
if (ColorDialog1->Execute()){
  Canvas->Brush->Color = ColorDialog1->Color;
}
```

Nyní již můžeme kreslit různé tvary, měnit jejich barvy, zadávat výplňové vzory apod. Vyzkoušejte.

5. V dalším kroku přidáme stavový řádek. Přestože můžeme k vytvoření stavového řádku využít komponentu **Panel**, je jednodušší použít přímo komponentu **StatusBar**. Tato komponenta nám umožní rozdělit stavový řádek na několik textových oblastí. Pro přidání stavového řádku k aplikaci, umístíme komponentu **StatusBar** na formulář, použijeme Editor stavového řádku k vytvoření stavových panelů na stavovém řádku a přidáme kód pro aktualizaci jednotlivých stavových panelů. Stavový řádek je většinou zobrazován na spodním okraji formuláře. Komponenta **StatusBar** má automaticky nastavenu svou vlastnost **Align** na *alBottom*. Již přidaný panel stavového řádku můžeme rozdělit na separátní stavové panely (není nutno složitě formátovat text na jednom panelu). Můžeme také určit, jak zarovnávat text na jednotlivých panelech. V naši aplikaci použijeme stavový řádek k zobrazování souřadnic počátku každého prvku a souřadnic aktuální pozice myši. V naši aplikaci přidáme na formulář komponentu **StatusBar**, nazveme ji *StatusBar* a rozdělíme ji na dva panely. Dále přidáme kód k obsluhám událostí, které aktualizují stavový řádek. Následuje text změněných obsluh událostí:

```
void __fastcall TForm1::FormMouseDown(TObject *Sender,
                TMouseButton Button, TShiftState Shift, int X, int Y)
{
  Kreslit = True;
  Canvas->MoveTo(X, Y);
  Pocatek = Point(X, Y);
```

```
  Presun = Pocatek;
  TVarRec tempvar[2] = {X, Y};
  StatusBar->Panels->Items[0]->Text = Format("Počátek: (%d, %d)",
                                              tempvar, 2);
}
void __fastcall TForm1::FormMouseMove(TObject *Sender, TShiftState Shift,
                                      int X, int Y)
{
  if (Kreslit){
    KresliTvar (Pocatek, Presun, pmNotXor);
    Presun = Point(X, Y);
    KresliTvar (Pocatek, Presun, pmNotXor);
  }
  TVarRec tempvar[2] = {X, Y};
  StatusBar->Panels->Items[1]->Text = Format("Aktuální: (%d, %d)",
                                              tempvar, 2);
}
```

6. Čas, kdy chceme kreslit přímo na formulář již minul. Mnohem častěji aplikace kreslí na bitové mapy, neboť bitové mapy jsou velmi flexibilní pro operace (jako je kopírování, tisk nebo uložení). Komponenta Image je komponenta, která může obsahovat bitovou mapu a umožňuje snadné vložení jedné nebo více bitových map na formulář. Bitová mapa také nemusí mít stejnou velikost jako formulář; může být menší nebo větší. V dalším kroku vývoje naší aplikace tedy použijeme komponentu Image.

   Často aplikace potřebuje zobrazovat více informací, než se vejde do jisté oblasti. Některé ovladače (např. okna seznamů) mohou automaticky rolovat svým obsahem. Ale jiné ovladače (včetně samotného formuláře) také poskytují schopnost rolování. Builder obsluhuje tyto posuvné oblasti ovladačem nazývaným *posuvné okno*. Posuvné okno se podobá panelu, který může obsahovat jiné ovladače, ale posuvné okno je normálně neviditelné. Jestliže ovladač obsažený v posuvném okně nemůže být ve viditelné oblasti posuvného okna, pak je automaticky zobrazen jeden nebo dva posuvníky, umožňující přesunout ovladač do zobrazované oblasti. K vytvoření posuvné oblasti, umístíme ovladač **ScrollBox** na formulář a nastavíme jeho meze na oblast, se kterou chceme rolovat (často to provedeme pomocí vlastnosti **Align**). V naši aplikaci vytvoříme posuvnou oblast zabírající celou plochu formuláře mezi paletou nástrojů a stavovým řádkem. Umístíme komponentu **ScrollBox** na formulář a nastavíme její vlastnost **Align** na *alClient*.

   Pomocí komponenty **Image** specifikujeme oblast na formuláři, která může obsahovat objekt obrázku (např. bitovou mapu nebo metasoubor). Velikost obrázku lze nastavit manuálně nebo využít ovladač **Image** k udržování velikosti svého obrázku při běhu aplikace. Ovladač **Image** můžeme umístit kdekoliv na formulář. Jestliže předpokládáme že tento ovladač bude schopen měnit svou velikost, pak nastavíme umístění jeho levého horního rohu. V naši aplikaci přidáme na formulář komponentu **Image** a nastavíme tyto její vlastnosti: **Name** na *Image*, **AutoSize** na *True*, **Left** na 0, **Top** na 0, **Width** na 200 a **Height** na 200. Když umístíme ovladač **Image**, je bez obrázku. Jestliže má obsahovat nějaký obrázek, můžeme při návrhu nastavit jeho vlastnost **Picture**. Ovladač také může zavést svůj obrázek ze souboru při běhu aplikace. Jestliže potřebujeme prázdnou bitovou mapu pro kreslení, můžeme ji vytvořit při běhu aplikace. K vytvoření prázdné bitové mapy při spuštění aplikace vytvoříme obsluhu události **OnCreate** pro formulář a vytvořený objekt bitové mapy přiřadíme vlastnosti **Picture.Graphics** ovladače **Image**. V naši aplikaci bude tato obsluha vypadat takto:

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  Graphics::TBitmap *Bitmap;
  Bitmap = new Graphics::TBitmap();
  Bitmap->Width = 200;
  Bitmap->Height = 200;
  Image->Picture->Graphic = Bitmap;
}
```

   Přiřazení bitové mapy vlastnosti **Graphic** obrázku vytváří vzájemný vztah bitové mapy a objektu obrázku. Bitovou mapu tedy není nutno rušit, zruší ji objekt obrázku. Můžeme přiřadit objektu obrázku další bitovou mapu a obrázek uvolní starou bitovou mapu a převezme řízení nové. Jestliže nyní spustíme aplikaci, vidíme na klientské oblasti formuláře bílou oblast reprezentující bitovou mapu. Jestliže klientská oblast okna nemůže zobrazit celý obrázek, jsou automaticky zobrazeny posuvníky pomocí nichž lze zobrazit zbytek obrázku. Obrázek ale nelze kreslit, neboť kreslení stále probíhá na formuláři, který je nyní překryt posuvným oknem a komponentou **Image**. Pro kreslení je nutno nyní použít plátno ovladače **Image** namísto plátna formuláře. Nejsnadněji to provedeme změnou všech výskytů „**Canvas**“ na „**Image->Canvas**“ v naší programové jednotce přiřazené k formuláři. Dále musíme přiřadit obsluhy událostí myši odpovídajícím událostem

ovladače **Image**. V Inspektoru objektu zobrazíme události objektu **Image** a odpovídajícím událostem přiřadíme již existující obsluhy událostí formuláře **FormMouseDown**, **FormMouseMove** a **FormMouseUp**. Původní obsluhy pro formulář můžeme zrušit, ale není to nutné. Nyní po spuštění aplikace již může kreslit, ale pouze na ovladači **Image**. Můžeme také skrýt nebo zobrazit paletu pera nebo paletu štětce a rolovat obrázkem.

7. V dalším kroku se budeme zabývat vytvořením nabídky pro naši aplikaci. Na formulář přidáme komponentu **MainMenu** a vytvoříme nabídku podle následující tabulky:

&Soubor	Úpr&avy
&Nový	&Vyjmout
&Otevřít ...	&Kopírovat
&Uložit	V&ložit
Uložit j&ako ...	
&Tisk	

&Konec	

   Nyní, když máme nabídku, můžeme vytvořit obsluhu události *OnClick* pro volbu **Soubor | Konec**. Obsluha bude tvořena příkazem *Close();*. Aplikaci můžeme spustit a vyzkoušet.

8. Tisk obrázku z aplikace Builderu je velmi jednoduchý. Musíme přidat hlavičkový soubor *vcl/printers.hpp* do programové jednotky formuláře, která bude tisknout. Tento hlavičkový soubor deklaruje objekt nazvaný *Printer*, který má plátno reprezentující tištěnou stranu. Obrázek vytiskneme kopírováním obrázku na plátno tiskárny. V naši aplikaci vytvoříme následující obsluhu události **OnClick** pro volbu **Soubor | Tisk**:

```
void __fastcall TForm1::Tisk1Click(TObject *Sender)
{
  Printer()->BeginDoc();
  Printer()->Canvas->CopyRect(Image->ClientRect, Image->Canvas,
                              Image->ClientRect);
  Printer()->EndDoc();
}
```

9. Obrázky existují pouze při běhu aplikace. Často chceme použít stejný obrázek nebo chceme uložit vytvořený obrázek pro pozdější použití. Komponenta **Image** usnadňuje zavádění obrázku a jejich ukládání. Princip ukládání a zavádění grafických souborů je stejný jako u jiných souborů. Na formulář tedy přidáme komponenty **OpenDialog** a **SaveDialog** a k objektu typu formuláře (do veřejné části deklarace) přidáme položku **AktualniSoubor** typu *AnsiString*. K zavedení grafického souboru do ovladače **Image** volíme metodu **LoadFromFile** objektu **Picture** ovladače **Image**. Vytvoříme tedy následující obsluhu události **OnClick** volby **Soubor | Otevřít**:

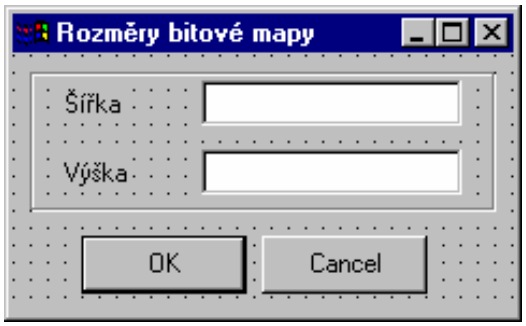
```
void __fastcall TForm1::Otevit1Click(TObject *Sender)
{
  if (OpenDialog1->Execute()){
    AktualniSoubor = OpenDialog1->FileName;
    Image->Picture->LoadFromFile(AktualniSoubor);
  }
}
```

   Když vytvoříme nebo modifikujeme obrázek, často jej chceme uložit do souboru pro pozdější použití. Objekt obrázku může ukládat grafiku v několika formátech a vývojář aplikace může vytvářet a registrovat vlastní formáty grafických souborů, které objekt obrázku pak může použít. K uložení obsahu ovladače **Image** do souboru voláme metodu **SaveToFile** objektu **Picture** ovladače **Image**. V naši aplikaci vytvoříme následující obsluhy událostí **OnClick** voleb **Soubor | Uložit** a **Soubor | Uložit jako**:

```
void __fastcall TForm1::Uloitjako1Click(TObject *Sender)
{
  if (SaveDialog1->Execute()){
    AktualniSoubor = SaveDialog1->FileName;
    Uloit1Click(Sender);
  }
}
void __fastcall TForm1::Uloit1Click(TObject *Sender)
{
  if (AktualniSoubor != EmptyStr){
    Image->Picture->SaveToFile(AktualniSoubor);
  }
  else{
    Uloitjako1Click (Sender);
```

```
  }
}
```

Obrázek v ovladači můžeme kdykoli při běhu aplikace nahradit jiným obrázkem. Jestliže přiřadíme novou grafiku k obrázku, který již má grafiku, pak nová grafika nahradí existující. Vytváření nové grafiky je stejný proces jako jsme použili při vytváření inicializační grafiky (obsluha události **OnCreate**), ale musíme dát uživateli možnost zvolit i jinou než implicitní velikost obrázku. To snadno provedeme pomocí dialogového okna. Vytvoříme vlastní dialogové okno (podle následujícího obrázku), nazveme jej *NovyBMPForm*, editační ovladače nazveme *SirkaEdit* a *VyskaEdit* a uložíme jej do BMPDlg.CPP.



Musíme také přidat hlavičkový soubor *BMPDlg* do jednotky hlavního formuláře a vytvoříme následující obsluhu události **OnClick** pro volbu **Soubor | Nový**:

```
void __fastcall TForm1::Nov1Click(TObject *Sender)
{
  Graphics::TBitmap *Bitmap;
  NovyBMPForm->ActiveControl = NovyBMPForm->SirkaEdit;
  NovyBMPForm->SirkaEdit->Text=IntToStr(Image->Picture->Graphic->Width);
  NovyBMPForm->VyskaEdit->Text=IntToStr(Image->Picture->Graphic->Height);
  if (NovyBMPForm->ShowModal() != IDCANCEL){
    Bitmap = new Graphics::TBitmap();
    Bitmap->Width = StrToInt(NovyBMPForm->SirkaEdit->Text);
    Bitmap->Height = StrToInt(NovyBMPForm->VyskaEdit->Text);
    Image->Picture->Graphic = Bitmap;
    AktualniSoubor = EmptyStr;
  }
}
```

Je důležité pochopit, že přiřazení nové bitové mapy k vlastnosti **Graphic** objektu obrázku, způsobí zrušení existující bitové mapy a vytvoření vzájemného vztahu s novou.

10. Schránku Windows můžeme použít pro kopírování a vkládání grafiky ve své aplikaci nebo k výměně grafiky s jinou aplikací. Objekt schránky Builderu pracuje s různými typy informací, včetně grafických. Dříve než můžeme použít objekt schránky ve své aplikaci, musíme přidat hlavičkový soubor *Clipbrd.hpp* do jednotky, ve které chceme schránku používat. Můžeme kopírovat libovolný obrázek, včetně obsahu ovladače **Image** do schránky. Po vložení do schránky je obrázek přístupný pro všechny aplikace Windows. Pro kopírování obrázku do schránky přiřadíme obrázek k objektu schránky použitím metody **Assign**. V naši aplikaci vytvoříme následující obsluhu události **OnClick** volby **Úpravy | Kopírovat**.

```
void __fastcall TForm1::Koprovat1Click(TObject *Sender)
{
  Clipboard()->Assign(Image->Picture);
}
```

Vyjmutí grafiky do schránky se podobá kopírování, s tím, že je ještě nutno zrušit kopírovanou grafiku. Nejprve vytvoříme kopii ve schránce a potom zrušíme originál. Vytvoříme tedy následující obsluhu události **OnClick** pro volbu **Úpravy | Vyjmout**:

```
void __fastcall TForm1::Vyjmout1Click(TObject *Sender)
{
  TRect ARect;
  Koprovat1Click(Sender);
  Image->Canvas->CopyMode = cmWhiteness;
  ARect = Rect(0, 0, Image->Width, Image->Height);
  Image->Canvas->CopyRect(ARect, Image->Canvas, ARect);
  Image->Canvas->CopyMode = cmSrcCopy;
}
```

Jestliže schránka obsahuje bitovou mapu, můžeme ji vložit do libovolného objektu obrázku, včetně ovladače **Image** nebo samotného formuláře. Pro vložení grafiky ze schránky, voláme metodu schránky *HasFormat*

k zjištění, zda schránka obsahuje grafiku (pro test na grafiku předáme metodě parametr CF_BITMAP) a pokud schránka grafiku obsahuje, pak ji přiřadíme. V naši aplikaci vytvoříme následující obsluhu události **OnClick** volby **Úpravy | Vložit**:

```
void __fastcall TForm1::Paste1Click(TObject *Sender)
{
  Graphics::TBitmap *Bitmap;
  if (Clipboard()->HasFormat(CF_BITMAP)){
    Bitmap = new Graphics::TBitmap();
    try{
      Bitmap->Assign(Clipboard());
      Image->Canvas->Draw(0, 0, Bitmap);
      delete Bitmap;
    }
    catch(...){
      delete Bitmap;
    }
  }
}
```

Nyní již se schránkou můžeme pracovat běžným způsobem a lze tedy ji využít pro přenos obrázků mezi různými aplikacemi. Naše aplikace je tedy hotova.

11. Pokuste se aplikaci grafického editoru doplnit nějakou další činností. Vytvořte pro tuto aplikaci také nápovědu.
12. Kreslení za běhu aplikace si ukážeme ještě na jedné jednoduché aplikaci. Začneme s vývojem nové aplikace. U formuláře nastavíme vlastnost **Color** na *clBlack*. Na formulář dále přidáme komponentu **Timer**, kde **Interval** nastavíme na 500. Před deklaraci třídy formuláře do hlavičkového souboru formuláře umístíme tyto deklarace:

```
const int MaxBodu = 15;
struct TRBod {
  float X, Y;
};
```

a do soukromé části deklarace třídy formuláře přidáme:

```
TRBod Body[MaxBodu];
float Natoceni;
int PocetBodu;
void __fastcall RotaceBodu();
```

Metoda *RotaceBodu* bude tvořena příkazy:

```
void __fastcall TForm1::RotaceBodu()
{
  const float M_2PI = 2 * M_PI;
  float KrokUhlu = M_2PI / PocetBodu;
  Natoceni += M_PI / 32;
  if (Natoceni > KrokUhlu) Natoceni -= KrokUhlu;
  int i;
  float j;
  for (i = 0, j = Natoceni; i < PocetBodu; i++, j += KrokUhlu) {
    Body[i].X = cos(j);
    Body[i].Y = sin(j);
  }
}
```

K formuláři vytvoříme obsluhu události **OnCreate** s příkazy:

```
Canvas->Pen->Color = clTeal;
Natoceni = 0;
PocetBodu = MaxBodu;
RotaceBodu ();
```

Dále vytvoříme obsluhu **OnPaint** formuláře. Zde budou příkazy:

```
int StredX = ClientWidth / 2;
int StredY = ClientHeight / 2;
int Polomer = min(StredY, StredX);
Canvas->Ellipse(0, 0, Polomer*2, Polomer*2);
int i,j;
for (i = 0; i < PocetBodu; i++) {
  for (j = i + 1; j < PocetBodu; j++) {
```

```
    Canvas->MoveTo(Polomer + floor(Body[i].X * Polomer),
                   Polomer + floor(Body[i].Y * Polomer));
    Canvas->LineTo(Polomer + floor(Body[j].X * Polomer),
                   Polomer + floor(Body[j].Y * Polomer));
  }
}
```

Obsluhu **OnResize** formuláře tvoří příkaz:

```
Invalidate();
```

Zbývá ještě vytvořit obsluhu **OnTimer** časovače. Zde budou příkazy:

```
RotaceBodu();
Invalidate();
```

Tím je naše aplikace hotova. Vyzkoušejte co dělá a snažte se pochopit význam jednotlivých metod.

13. Dále se seznámíme s používáním komponenty **HeaderControl**. Začneme s vývojem nové aplikace. Na horní okraj formuláře umístíme komponentu **HeaderControl** a nastavíme u ní vlastnost **Sections** na tři sekce s texty *Červená*, *Zelená* a *Modrá* (šířku jednotlivých sekcí nastavíme na 125, minimální šířku na 50, maximální šířku na 250 a zarovnávání na centrování). Na formulář dále vložíme vedle sebe tři komponenty **Shape** tak aby zaplnily šířku jednotlivých sloupců hlavičky, vlastnost **Shape** u nich nastavíme na *stEllipse* a barvu výplně změníme podle textu v hlavičce. Dále vytvoříme obsluhu události **OnSectionTrack** ovladače **HeaderControl**, která bude tvořena příkazy:

```
Section->Width = Width;
Shape1->Width = HeaderControl->Sections->Items[0]->Width;
Shape2->Width = HeaderControl->Sections->Items[1]->Width;
Shape2->Left = HeaderControl->Sections->Items[1]->Left;
Shape3->Width = HeaderControl->Sections->Items[2]->Width;
Shape3->Left = HeaderControl->Sections->Items[2]->Left;
```

Tím je aplikace hotova, můžeme ji vyzkoušet. Myší lze přetahovat dělící čáry v hlavičce a tím měnit šířku zobrazené elipsy ve sloupci jehož šířku měníme.

14. V této kapitole se budeme ještě zabývat vytvářením hodin. Na formulář umístíme komponenty **Timer** a **Panel** (**Panel** zvětšíme na celou plochu formuláře a také zvětšíme písmo jeho titulku). Vytvoříme ještě obsluhu události **OnTimer** časovače. Bude tvořena příkazem:

```
Panel1->Caption = TimeToStr(Time());
```

Funkce *Time* vrací objekt typu **TDateTime**, který obsahuje aktuální čas a funkce *StrToTime* jej převede na řetězec. Tím jsou nejednodušší (digitální) hodiny hotovy. Tvar výpisu je určen nastavením národnostních zvyklostí ve Windows.

15. V další aplikaci se pokusíme vytvořit analogové hodiny. Formulář, který potřebujeme pro ručičkové hodiny je ještě jednodušší. Obsahuje pouze časovač. Většina kódu je v obsluze události **OnPaint**. Protože potřebujeme vykreslovat tři různé hodinové ručičky přidáme do formuláře veřejnou metodu, kterou budeme kreslit jednotlivé ručičky. Bude to funkce:

```
void __fastcall TForm1::KresliRucicku(int XStred, int YStred,
                        int Polomer, int ZpPolomer, double Uhel)
{
  Canvas->MoveTo(
    XStred-floor(ZpPolomer*cos(Uhel)),YStred-floor(ZpPolomer*sin(Uhel)));
  Canvas->LineTo(
    XStred+floor(Polomer*cos(Uhel)),YStred+floor(Polomer*sin(Uhel)));
}
```

Do deklarace formuláře ještě vložíme následující soukromé položky:

```
unsigned short int Hodina, Minuta, Sekunda;
int XStred, YStred, Polomer;
```

Obsluha události **OnPaint** formuláře je tvořena následujícími příkazy:

```
double Uhel;
int I, X, Y, Velikost;
XStred = ClientWidth / 2;
YStred = ClientHeight / 2;
if (XStred > YStred) Polomer = YStred - 10;
else Polomer = XStred - 10;
Canvas->Pen->Color = clYellow;
Canvas->Brush->Color = clYellow;
Velikost = Polomer / 50 + 1;
for (I = 0; I < 12; I++){
```

```
    Uhel = 2 * M_PI * (I + 9) / 12;
    X = XStred - floor(Polomer * cos(Uhel));
    Y = YStred - floor(Polomer * sin(Uhel));
    Canvas->Ellipse(X-Velikost, Y-Velikost, X+Velikost, Y+Velikost);
  }
  Canvas->Pen->Width = 2;
  Canvas->Pen->Color = clBlue;
  Uhel = 2 * M_PI * (Minuta+45) / 60;
  KresliRucicku(XStred, YStred, Polomer * 90 / 100, 0, Uhel);
  Uhel = 2 * M_PI * (Hodina + 9 + Minuta / 60) / 12;
  KresliRucicku(XStred, YStred, Polomer * 70 / 100, 0, Uhel);
  Canvas->Pen->Width = 1;
  Canvas->Pen->Color = clRed;
  Uhel = 2 * M_PI * (Sekunda + 45) / 60;
  KresliRucicku(XStred, YStred, Polomer, Polomer * 30 / 100, Uhel);
```

Zbývá ještě vytvořit několik obsluh událostí. Obsluha **OnTimer** časovače je tvořena příkazy:

```
unsigned short SetinySec;
DecodeTime(Time(), Hodina, Minuta, Sekunda, SetinySec);
Refresh();
```

Obsluhu **OnResize** formuláře tvoří příkaz: `Refresh();` Jestliže vytvoříme ještě obsluhu události **OnCreate** formuláře s následujícím příkazem, pak hodiny budou ukazovat správný čas ihned od zobrazení.

```
Timer1Timer(this);
```

Vytvořte tyto hodiny.

16. Předchozí program sice funguje, ale k dokonalosti mu ještě něco chybí. Obrázek hodin je nestabilní; příliš bliká. Ve skutečnosti je totiž každou sekundu celý obsah formuláře vymazán a znovu vykreslen. Pokuste se tento program vylepšit tak, že smažete pouze starou sekundovou ručičku a potom ji nakreslete znova na novém místě.
17. Chceme-li uložit informaci o stavu aplikace, abychom ji mohli při dalším spuštění programu obnovit, můžeme ji uložit do souboru, a to v takovém formátu, jaký uznáme za vhodný. Nicméně Windows zajišťují explicitní podporu pro inicializační soubory (s příponou INI). Každá aplikace může mít svůj vlastní INI soubor a zapisovat do něj řetězce, čísla nebo hodnoty typu bool. Stejné hodnoty je možné snadno číst. Builder obsahuje třídu **TIniFile**, s jejíž pomocí můžeme manipulovat s INI soubory. Vytvoříme objekt této třídy, přiřadíme mu soubor a můžeme z něj informace jak číst, tak je do něj zapisovat. Pro vytvoření tohoto objektu je nutné zavolat konstruktor, kterému předáme jako parametr název souboru.

    Existují dvě možnosti umístění INI souboru na disku. Jednak je možné uložit jej do adresáře aplikace a konstruktoru předat úplnou cestu k němu. Druhým běžnějším řešením je uložit inicializační soubor do adresáře Windows. V tomto případě zadáváme pouze název souboru.

    INI soubory jsou rozděleny na sekce označené názvem uzavřeným v hranatých závorkách. Každá sekce obsahuje několik prvků tří možných druhů: řetězec, celé číslo nebo logická hodnota. Třída **TIniFile** obsahuje metody pro čtení každého z těchto typů a jsou zde též metody pro jejich zápis. V metodách určených ke čtení můžeme specifikovat implicitní hodnotu, která bude použita v případě, že odpovídající záznam v INI souboru neexistuje. Využití INI souborů si ukážeme v následující aplikaci. Formulář aplikace je prázdný a jsou zde pouze obsluhy událostí **OnCreate** a **OnClose** formuláře. Do INI souboru budeme ukládat umístění a velikost našeho formuláře a při dalším spuštění toto umístění a velikost použijeme. Do soukromé části deklarace formuláře umístíme (do hlavičkového souboru formuláře vložíme `#include <vcl/inifiles.hpp>`):

```
TIniFile *IniFile;
```

    Obsluha **OnCreate** je tvořena příkazy:

```
int Stav;
IniFile = new TIniFile("Project1.ini");
Stav = IniFile->ReadInteger("Form1","Status", 0);
if (Stav != 0) {
  Top = IniFile->ReadInteger("Form1","Top", 0);
  Left = IniFile->ReadInteger("Form1","Left", 0);
  Width = IniFile->ReadInteger("Form1","Width", 0);
  Height = IniFile->ReadInteger("Form1","Height", 0);
  switch (Stav) {
    case 2: WindowState = wsMinimized; break;
    case 3: WindowState = wsMaximized; break;
  }
}
```

a obsluhu **OnClose** tvoří:

```
int Stav;
if (Application->MessageBox("Ulo□it aktuální stav formuláøe?",
               "Dotaz", MB_YESNO | MB_ICONQUESTION ) == IDYES) {
  switch (WindowState) {
      case wsNormal:
        IniFile->WriteInteger("Form1","Top", Top);
        IniFile->WriteInteger("Form1","Left", Left);
        IniFile->WriteInteger("Form1","Width", Width);
        IniFile->WriteInteger("Form1","Height", Height);
        Stav = 1;
        break;
      case wsMinimized: Stav = 2; break;
      case wsMaximized: Stav = 3; break;
  }
  if (!Active) Stav = 2;
  IniFile->WriteInteger("Form1","Status", Stav);
}
delete IniFile;
```

Tím je naše aplikace hotova a můžeme ji vyzkoušet. Prohlédněte si také v nějakém textovém editoru vytvořený INI soubor. Pokuste se pochopit, jak tato aplikace pracuje.

8. Příklad správce souborů

1. V této kapitole se budeme zabývat postupným vývojem aplikace jednoduchého správce souborů. Vývoj naši aplikace budeme provádět v těchto krocích: Navrhneme formulář správce souborů, propojíme ovladače (aplikace používá tři různé ovladače k zobrazení přípustných diskových jednotek, adresářů a souborů), vytvoříme vlastní zobrazovací ovladač, zajistíme práci se soubory a vytvoříme alternativní způsob práce se soubory pomocí myši. Nejprve provedeme návrh formuláře aplikace správce souborů. Začneme vytvořením nového projektu. Použitím Inspektora objektů nastavíme následující vlastnosti pro hlavní formulář projektu: **Caption** nastavíme na *Správce souborů*, **Name** na *FMForm* a **Position** na *poDefault*. Nyní přidáme na formulář ovladače. Protože umístění každého z nich závisí na umístění ostatních je nutno jejich umístění a nastavení jejich vlastností zadávat v pořadí uvedeném v následující tabulce. Komponenty **DirectoryPanel** a **FilePanel** jsou umístěny na **StatusBar**:

Komponenta	Vlastnost	Hodnota
Panel	Align	alBottom
	Caption	necháme prázdný
	Name	StatusBar
	BevelOuter	bvNone
Panel	Align	alLeft
	Caption	necháme prázdný
	Name	DirectoryPanel
	BevelInner	bvLowered
	BevelWidth	2
Panel	Align	alClient
	Caption	necháme prázdný
	Name	FilePanel
	BevelInner	bvLowered
	BevelWidth	2
TabSet	Align	alBottom
	Name	DriveTabSet
DirectoryListBox	Name	DirectoryOutline
	Align	alLeft
FileListBox	Align	alClient
	Name	FileList
	ShowGlyphs	True

Když máme přidány všechny komponenty a nastaveny jejich vlastnosti je vhodné projekt uložit. **Unit1** uložíme pod jménem *FMXWin* a **Project1** pod *FilManEx*. K projektu dále přidáme následující programovou jednotku (nazveme ji *FMXUtils*). Obsahuje různé pomocné funkce pro práci se soubory (prostudujte si je a zjistěte co a jak dělají). Nejprve je uveden hlavičkový soubor a následuje jeho programová jednotka:

```
#ifndef FmxUtilsHPP
#define FmxUtilsHPP
extern void __fastcall CopyFile(const AnsiString FileName,
                                const AnsiString DestName);
extern void __fastcall MoveFile(const AnsiString FileName,
                                const AnsiString DestName);
extern long __fastcall GetFileSize(const AnsiString FileName);
extern TDateTime __fastcall FileDateTime(const AnsiString FileName);
extern bool __fastcall HasAttr(const AnsiString FileName, unsigned short Attr);
extern int __fastcall ExecuteFile(const AnsiString FileName,
  const AnsiString Params, const AnsiString DefaultDir, int ShowCmd);
#endif

#include <vcl\vcl.h>
#pragma hdrstop
#include <shellapi.h>
#include "FMXUtils.h"
void __fastcall CopyFile(AnsiString FileName,AnsiString DestName)
{
  bool ckcopy;
  AnsiString Destination;
  char FName[255];
  GetFileTitle(FileName.c_str(),FName,255);
  Destination = ExpandFileName(DestName);
  if(HasAttr(Destination,faDirectory))
    Destination=Destination+FName;
  ckcopy= CopyFile(FileName.c_str(),Destination.c_str(),false);
  if(!ckcopy)
    ShowMessage("Zadejte jméno cílového souboru");
}
void __fastcall MoveFile(const AnsiString FileName,const AnsiString DestName)
{
  AnsiString Destination;
  char FName[255];
  bool ckmove;
  Destination=ExpandFileName(DestName);
  GetFileTitle(FileName.c_str(),FName,255);
  if(HasAttr(FileName, faReadOnly)) {
      char buffer[255];
      sprintf(buffer, "Chyba: Nemohu přesunout soubor '%s'.",
              FileName.c_str());
      ShowMessage(buffer);
      return;
  }
  if (HasAttr(Destination,faDirectory))
      Destination=Destination+AnsiString(FName);
  ckmove= MoveFile(FileName.c_str(), Destination.c_str());
  if(!ckmove)
    ShowMessage("Zadejte jméno cílového souboru");
}
long __fastcall GetFileSize(const AnsiString FileName)
{
  TSearchRec SearchRec;
  if(FindFirst(ExpandFileName(FileName), faAnyFile, SearchRec)==0)
    return SearchRec.Size;
  else return -1;
}
TDateTime __fastcall FileDateTime(const AnsiString FileName)
{
  return (FileDateToDateTime(FileAge(FileName)));
}
bool __fastcall HasAttr(const AnsiString FileName,const unsigned short Attr)
{
  int attribtest;
  attribtest=FileGetAttr(FileName);
```

```
  if(attribtest & Attr) return true;
  else return false;
}
int __fastcall ExecuteFile(const AnsiString FileName,
    const AnsiString Params, const AnsiString DefaultDir, int ShowCmd)
{
  char zFileName[79], zParams[79], zDir[79];
  return (int) ShellExecute(Application->MainForm->Handle, NULL,
                            strcpy(zFileName, FileName.c_str()),
                            strcpy(zParams, Params.c_str()),
                            strcpy(zDir, DefaultDir.c_str()), ShowCmd);
}
```

K našemu formuláři přidáme i nabídku. Na formulář přidáme komponentu **MainMenu** a pomocí Návrháře nabídky vytvoříme nabídku podle následující tabulky (jsou zde uvedeny i zkracovací klávesy):

```
&Soubor
-----------------------
&Otevřít        Enter
&Přesunout      F7
Kopí&rovat      F8
&Zrušit         Del
Přej&menovat
&Atributy       Alt+Enter
------------
&Konec
```

Nyní již můžeme vytvořit obsluhu události **OnClick** pro volbu **Soubor | Konec**, která uzavírá hlavní formulář aplikace. Vytvořte ji sami.

2. Po přidání komponenty **TabSet** na formulář, se můžeme zabývat zprovozněním tohoto ovladače. Tato komponenta vytváří záložky pro každý prvek v jeho seznamu záložek v závislosti na vlastnosti **Tabs**. Implicitně tento seznam je prázdný. Seznam sice můžeme editovat během návrhu, ale budeme jej vytvářet až při běhu aplikace. Po zprovoznění tohoto ovladače jej musíme propojit s ovladačem **DirectoryOutline** a následně s **FileList**.

   API Windows poskytuje funkci *GetDriveType*, která vrací informaci o typu specifikované jednotky. K určení zda jednotka je přípustná, předáme nulou ukončený řetězec obsahující kořenový adresář jednotky, funkci *GetDriveType*. Vrácená hodnota indikuje typ jednotky: hodnota větší než 1 indikuje přípustnou jednotku; jiná hodnota indikuje nepřípustnou jednotku. Následující kód tvoří obsluhu události **OnCreate** formuláře a slouží k vytvoření seznamu záložek při vytvoření formuláře. Seznam záložek obsahuje přípustné jednotky v systému.

```
static char * Jednotka[]={"a:\\","b:\\","c:\\","d:\\","e:\\","f:\\",
  "g:\\","h:\\","i:\\","j:\\","k:\\","l:\\","m:\\","n:\\","o:\\","p:\\",
  "q:\\","r:\\","s:\\","t:\\","u:\\","v:\\","w:\\","x:\\","y:\\","z:\\"};
int  PridanyIndex;
for(int x =0; x <= 25; x++ )
  if(GetDriveType(Jednotka[x]) > 1) {
    PridanyIndex = DriveTabSet->Tabs->Add(String(Jednotka[x]));
    if (toupper(*Jednotka[x]) == FileList->Drive)
      DriveTabSet->TabIndex = PridanyIndex;
  }
```

   Tato obsluha vloží písmena přípustných jednotek na záložky. Později se ještě budeme zabývat zobrazením typu jednotky. Nyní, když máme ovladače reprezentující diskové jednotky, adresáře a soubory, je nutno je propojit. Když zvolíme jinou diskovou jednotku, chceme zobrazit její adresářovou strukturu a seznam souborů vybraného adresáře. K propojení ovladačů, musíme vytvořit obsluhy událostí reagující na výběr záložky a změnu adresáře. Když uživatel kliknutím (nebo pomocí klávesnice) vybere záložku komponenty **TabSet**, komponenta generuje událost **OnClick**. Libovolný jiný ovladač, který závisí na nastavení záložky komponenty **TabSet** může použít její hodnotu k aktualizaci v reakci na tuto událost kliknutí. V našem případě **TabSet** obsahuje záložky pro přípustné diskové jednotky a kliknutí na některé záložce znamená změnu jednotky. Jiný ovladač, v našem případě adresářový seznam, musí reagovat na tuto změnu. Vytvoříme tedy následující obsluhu události **OnClick** pro komponentu nazvanou **DriveTabSet**, která nastaví vlastnost **Drive** v ovladači **DirectoryOutline** na první písmeno použité záložky:

```
DirectoryOutline->Drive= *((DriveTabSet->Tabs->Strings
                            [DriveTabSet->TabIndex]).c_str());
```

   Jestliže nyní vybereme záložku diskové jednotky, pak **DirectoryOutline** zobrazí adresářovou strukturu specifikované jednotky. Když uživatel vybere prvek v komponentě **DirectoryOutline** (kliknutím nebo pomocí

kurzorových kláves), pak komponenta generuje událost **OnClick**. Mnohem praktičtější je ale využívat událost **OnChange**, která indikuje, že něco v komponentě **DirectoryOutline** bylo změněno. Reakci na tuto událost můžeme využít k aktualizaci seznamu souborů. Následující kód aktualizuje jak seznam souborů tak i stavový řádek a to vždy při změně adresáře:

```
FileList->Directory = DirectoryOutline->Directory;
DirectoryPanel->Caption=DirectoryOutline->Directory;
```

Jakékoli změny v adresářovém stromu se projeví jednak zobrazením seznamu souborů vybraného adresáře a výpisem adresářové cesty k aktuálnímu adresáři na panelu stavového řádku. Když uživatel klikne na prvek v seznamu souborů, pak okno seznamu generuje událost **OnChange**. V našem případě využijeme tuto událost k zobrazení informací o vybraném souboru (na druhém panelu stavového řádku). Vytvoříme tedy tuto obsluhu události:

```
AnsiString JmenoSouboru;
if(FileList->ItemIndex>=0){
  char buffer[255];
  JmenoSouboru =FileList->Items->Strings[FileList->ItemIndex];
  sprintf(buffer, "%s  %d bytes", JmenoSouboru,
          GetFileSize(JmenoSouboru));
  FilePanel->Caption = buffer;
  if (GetFileAttributes(JmenoSouboru.c_str()) & FILE_ATTRIBUTE_DIRECTORY)
    VybranySoubor = false;
  else VybranySoubor = true;
}
else {
  FilePanel->Caption="";
  VybranySoubor = false;
}
```

Funkce *GetFileSize* je deklarována v programové jednotce *FMXUnit*. Do soukromé části deklarace formuláře umístíme:

```
bool VybranySoubor;
```

Změna vybraného souboru v seznamu souborů způsobí výpis jména a velikosti vybraného souboru na stavovém řádku.

3. Každý ovladač, který má vlastní variantu zobrazování má vlastnost nazvanou **Style**. Tato vlastnost určuje, zda ovladač používá implicitní zobrazování (nazvané standardní styl) nebo vlastní kreslení. Pro seznamy a kombinovaná okna je také volba vlastního kreslení nazvaná *Fixed* (každý prvek má stejnou šířku určenou vlastností **ItemHeight**) a *Variable* (prvky mají různou šířku). V naši aplikaci nastavíme vlastnost Style komponenty **TabSet** na *tsOwnerDraw*. Každý seznam řetězců Builderu může obsahovat seznam objektů. Do správce souborů přidáme bitové mapy indikující typ diskové jednotky. Požadované obrázky bitových map musíme přidat k aplikaci a potom tyto obrázky lze umisťovat do seznamu řetězců. Ovladač **Image** je ovladač obsahující grafický obrázek, např. bitovou mapu. Komponentu **Image** lze použít k zobrazení grafického obrázku na formuláři, ale můžeme jej také použít pro držení skrytých obrázků, které pak v aplikaci budeme používat. Lze je např. použít k uložení bitových map pro vlastní zobrazovací ovladač. V naši aplikaci přidáme na hlavní formulář tři komponenty **Image**, nastavíme jejich jména na *Floppy*, *Fixed* a *Network*, vlastnosti **Visible** u všech tří komponent nastavíme na False a vlastnosti **Picture** přiřadíme pomocí Inspektora objektů přes Editor obrázků vytvořené odpovídající bitové mapy. Když již máme grafické obrázky v aplikaci, můžeme je přiřadit k řetězcům v seznamu řetězců. Můžeme objekty obrázků přidávat současně s řetězci, nebo objekty obrázků přiřadit k již existujícím řetězcům. V naši aplikaci zjistíme existující disková zařízení a pro každé existující zařízení do seznamu přidáme řetězec současně s obrázkem reprezentujícím typ zařízení. Změníme tedy již existující obsluhu události **OnCreate** pro formulář a to takto:

```
static char * Jednotka[]={"a:\\","b:\\","c:\\","d:\\","e:\\","f:\\",
  "g:\\","h:\\","i:\\","j:\\","k:\\","l:\\","m:\\","n:\\","o:\\","p:\\",
  "q:\\","r:\\","s:\\","t:\\","u:\\","v:\\","w:\\","x:\\","y:\\","z:\\"};
int  PridanyIndex;
for(int x =0; x <= 25; x++ ) {
  switch(GetDriveType(Jednotka[x])) {
    case DRIVE_REMOVABLE:
    case DRIVE_CDROM:
    case DRIVE_RAMDISK:
      PridanyIndex=DriveTabSet->Tabs->AddObject(String(Jednotka[x]),
                                  Floppy->Picture->Graphic);
      break;
    case DRIVE_FIXED:
```

```
        PridanyIndex=DriveTabSet->Tabs->AddObject(String(Jednotka[x]),
                                  Fixed->Picture->Graphic);
        break;
      case DRIVE_REMOTE:
        PridanyIndex=DriveTabSet->Tabs->AddObject(String(Jednotka[x]),
                                  Network->Picture->Graphic);
        break;
    }
    if (toupper(*Jednotka[x]) == FileList->Drive)
      DriveTabSet->TabIndex = PridanyIndex;
}
```

Když nastavíme styl ovladače na vlastní kreslení, Windows již dále nezobrazuje ovladač na obrazovku. Místo toho generuje události pro každý viditelný prvek v ovladači. Naše aplikace tyto události zpracuje pro nakreslení prvků. Před vlastním zobrazením prvků v proměnném vlastním kreslícím ovladači, Windows generuje událost **OnMeasureItem**. Windows určují, jaká plocha bude potřebná k zobrazení prvku (obecně plocha, na které bude zobrazen text prvku aktuálním písmem). Naše aplikace může zpracovat tuto událost a změnit navrženou plochu. Např. chceme-li nahradit text prvku bitovou mapou, změníme plochu na velikost bitové mapy. Jestliže chceme bitovou mapu i text, zvětšíme požadovanou plochu tak, aby obsahovala obojí. K změně velikosti prvku pro vlastní kreslení, vytvoříme obsluhu události **OnMeasureItem**. Tato událost má dva důležité parametry: index prvku a velikost tohoto prvku. Velikost je parametr volaný odkazem a aplikace může jeho hodnotu zmenšovat nebo zvětšovat. Pozice následujícího prvku závisí na velikosti předchozího prvku. Změnu velikosti prvku lze využívat v komponentách seznamu, kombinovaného okna a **TabSet** (zde se používá událost **OnMeasureTab**). Není možno ji použít v komponentě mřížky (zde musí být velikost celého sloupce a řádku stejná). V naši aplikaci vytvoříme následující obsluhu události **OnMeasureTab** pro komponentu **TabSet**, která zvětší původní šířku záložky o šířku bitové mapy + 2:

```
int BitmapWidth;
BitmapWidth = ((Graphics::TBitmap *)
              (DriveTabSet->Tabs->Objects[Index]))->Width;
TabWidth=TabWidth+2+BitmapWidth;
```

Prvek z vlastnosti **Objects** v seznamu řetězců musíme přetypovat. **Objects** je vlastnost typu **TObject** a může tak obsahovat libovolného potomka **TObject**. Když získáváme objekty z pole, je potřeba je převést zpět na aktuální typ prvků. Jestliže aplikace požaduje kreslení nebo překreslení ovladače pomocí vlastního kreslení, Windows generuje kreslící událost pro každý viditelný prvek v ovladači. Jména těchto událostí začínají vždy **OnDraw**, např. **OnDrawItem**, **OnDrawTab** nebo **OnDrawCell**. Tato událost obsahuje parametry indikující index kresleného prvku, obdélník, ve kterém bude zobrazen a obvykle některé informace o stavu prvku. Pro naši aplikaci tedy vytvoříme následující obsluhu události **OnDrawTab** pro komponentu **TabSet**:

```
Graphics::TBitmap *Bitmap;
Bitmap = (Graphics::TBitmap *) (DriveTabSet->Tabs->Objects[Index]);
TabCanvas->Draw(R.Left, R.Top + 4, Bitmap);
TabCanvas->TextOut(R.Left + 2 + Bitmap->Width, R.Top + 2,
                   DriveTabSet->Tabs->Strings[Index].SubString(1,1));
```

Většina kreslících událostí nepředává jako parametr plátno objektu; běžně se při kreslení používá plátno ovladače. Protože **TabSet** má mezi prvky vložen oddělovač záložek, je předáno pro kreslený prvek speciální plátno jako parametr.

4. V knihovně běhu programu Builderu je zabudováno několik souborových operací. Funkce pro práci se soubory vyžadují zadání jména souboru, se kterým chceme pracovat. Mimo těchto knihovních funkcí budeme v naši aplikaci používat některé další; jsou uvedeny v programové jednotce *FMXUtils*. Práci se soubory budeme provádět jako reakci na volbu v nabídce. Protože připojíme funkce k prvkům nabídky, je vhodné zpřístupňovat tyto prvky pouze v případě vybraného souboru. Pro povolování a zakazování prvků v nabídce vytvoříme následující obsluhu události **OnClick** pro nabídku **Soubor**:

```
Otevt1->Enabled = VybranySoubor;
Zruit1->Enabled = VybranySoubor;
Koprovat1->Enabled = VybranySoubor;
Pesunout1->Enabled = VybranySoubor;
Pejmenovat1->Enabled = VybranySoubor;
Atributy1->Enabled = VybranySoubor;
```

Nyní, když uživatel otevře nabídku **Soubor**, aplikace povoluje nebo zakazuje všechny prvky této nabídky (mimo **Konec**) v závislosti na tom, zda máme v seznamu souborů vybraný soubor.

Zrušení souboru odstraní soubor z disku a zruší příslušnou položku v diskovém adresáři. Pro zrušení souboru, předáme jméno souboru funkci *DeleteFile*. *DeleteFile* vrací *true*, jestliže soubor byl zrušen, nebo *false*,

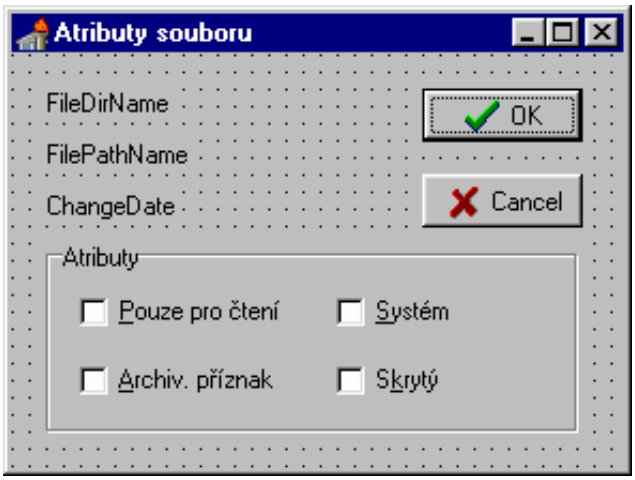
pokud jej nelze zrušit (jedná se např. o soubor určený pouze pro čtení). Následující kód zpracovává klinknutí na volbě **Soubor | Zrušit**:

```
if(MessageDlg("Zrušit " + FileList->FileName + "?", mtConfirmation,
              TMsgDlgButtons() << mbYes << mbNo, 0) == mrYes){
  if(DeleteFile(FileList->FileName.c_str()))
    FileList->Items->Delete(FileList->ItemIndex);
}
```

Každý soubor má několik atributů uložených operačním systémem jako bitové příznaky. Změna atributu souboru se provádí v těchto krocích: přečteme atributy souboru, změníme požadovaný atribut a uložíme atributy souboru. Pro přečtení atributů souboru, předáme jméno souboru funkci *FileGetAttr*. Vrácená hodnota typu *unsigned short* popisuje nastavené atributy. V našem příkladě otevřeme volbou **Soubor | Atributy** dialogové okno, ve kterém uživatel zjistí informace o souboru a kde také může atributy souboru měnit. Přidáme tedy k projektu nový formulář, přiřazenou programovou jednotku uložíme pod jménem *FAttrDlg* a vlastnosti nového formuláře nastavíme podle následující tabulky:

Vlastnost	Hodnota
Name	FileAttrDlg
Caption	Atributy souboru
Position	poScreenCenter
BorderIcons	[biSystemMenu]
BorderStyle	bsDialog

Podle následujícího obrázku na formulář přidáme další ovladače (jména tří komponent **Label** jsou uvedena na obrázku a značky nazveme *ReadOnly*, *Archive*, *System* a *Hidden*):



Pro nastavování atributů souboru je možno použít běžné bitové operace. Každý atribut má definované jméno. Např. nastavení atributu „Pouze pro čtení“ provedeme příkazem:

```
Atributy = Atributy | faReadOnly;
```

Pro nastavení souborových atributů, předáme jméno souboru a atributy funkci *FileSetAttr*. Následující kód čte souborové atributy do proměnné, nastaví značky v dialogovém okně na současné hodnoty atributů a dialogové okno provede. Jestliže uživatel některý z atributů změní, pak v případě uzavření dialogového okna pomocí tlačítka OK, jsou jejich hodnoty předány souboru (následující příkazy tvoří obsluhu volby **Soubor | Atributy** v nabídce):

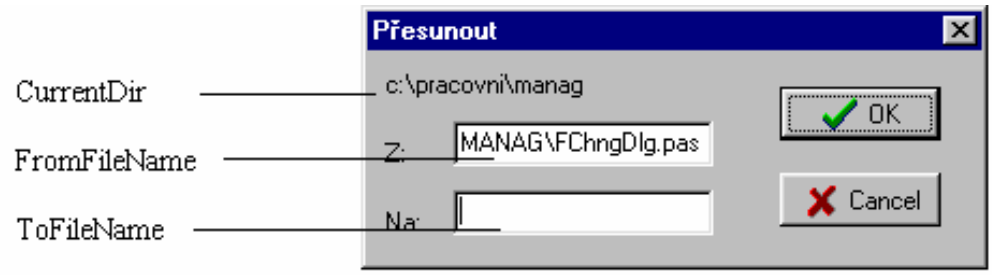
```
unsigned short  Atributy, NoveAtributy;
FileAttrDlg->FileDirName->Caption =
  FileList->Items->Strings[FileList->ItemIndex];
FileAttrDlg->FilePathName->Caption = FileList->Directory;
FileAttrDlg->ChangeDate->Caption =
  DateTimeToStr(FileDateTime(FileList->FileName));
Atributy = FileGetAttr(FileList->Items->Strings[FileList->ItemIndex]);
FileAttrDlg->ReadOnly->Checked = Atributy & faReadOnly;
FileAttrDlg->Archive->Checked = Atributy & faArchive;
FileAttrDlg->System->Checked = Atributy & faSysFile;
FileAttrDlg->Hidden->Checked = Atributy & faHidden;
if (FileAttrDlg->ShowModal()!= mrCancel){
  NoveAtributy = Atributy;
  if (FileAttrDlg->ReadOnly->Checked) NoveAtributy=NoveAtributy|faReadOnly;
  else NoveAtributy = NoveAtributy & ~faReadOnly;
  if (FileAttrDlg->Archive->Checked) NoveAtributy=NoveAtributy|faArchive;
```

```
  else NoveAtributy = NoveAtributy & ~faArchive;
  if (FileAttrDlg->System->Checked) NoveAtributy=NoveAtributy|faSysFile;
  else NoveAtributy = NoveAtributy & ~faSysFile;
  if (FileAttrDlg->Hidden->Checked) NoveAtributy=NoveAtributy|faHidden;
  else NoveAtributy = NoveAtributy  & ~faHidden;
  if (NoveAtributy != Atributy)
    FileSetAttr(FileAttrDlg->FileDirName->Caption, NoveAtributy);
}
```

Přesun, kopírování a přejmenování souborů jsou podobné operace, všechny vytvářejí soubor z jiného souboru. Liší se pouze tím, co zbude z původního souboru. Knihovna běhu programu poskytuje funkci *RenameFile* která provádí přejmenování. Programová jednotka *FMXUtils* poskytuje podobné procedury nazvané *MoveFile* a *CopyFile* pro další operace. Všechny tyto procedury přebírají dva řetězce jako své parametry: jméno původního souboru a jméno cílového souboru. Následující kód zobrazuje soukromou metodu nazvanou *ConfirmChange*, která zobrazí potvrzovací dialogové okno:

```
void __fastcall TFMForm::ConfirmChange(const AnsiString ACaption,
                          AnsiString FromFile, AnsiString ToFile)
{
  char buffer[700];
  sprintf(buffer,"%s %s na %s?",ACaption, FromFile, ToFile);
  if(MessageDlg(buffer, mtConfirmation,
                TMsgDlgButtons() << mbYes << mbNo, 0) == mrYes){
    if (ACaption == "Přesunout") MoveFile(FromFile, ToFile);
    else if (ACaption == "Kopírovat") CopyFile(FromFile, ToFile);
    else if (ACaption == "Přejmenovat") RenameFile(FromFile, ToFile);
    FileList->Update();
   }
}
```

Protože všechny tři operace jsou si podobné, může aplikace správce souborů sdílet většinu použitého kódu. Vytvoříme dialogové okno, ve kterém uživatel zadá původní jméno a cílové jméno a použijeme jej v těchto operacích. Na obrázku je toto okno zobrazeno a jsou zde pojmenovány jeho komponenty.



Formulář dialogového okna nazveme *ChangeDlg* a jeho jednotku uložíme do souboru *FChngDlg.CPP*. Po vytvoření dialogového okna jej můžeme otevírat z obsluhy události sdílené prvky nabídky **Přesunout**, **Kopírovat** a **Přejmenovat**. Na hlavním formuláři tedy dvojitě klikneme na komponentě **MainMenu** (ne na samotné nabídce **Soubor**), čím se otevře návrhář nabídky. V Návrháři nabídky zvolíme **Soubor | Přesunout**, v Inspektoru objektů vybereme stránku událostí, ve sloupci hodnot u události **OnClick** zapíšeme *FileChange* a stiskneme Enter. V Návrháři nabídky zvolíme **Soubor | Kopírovat** a nastavíme jeho ovladač **OnClick** na *FileChange*. Totéž provedeme pro **Soubor | Přejmenovat**. Obsluhu události *FileChange* doplníme takto (funkce *_c_exit* je definována v hlavičkovém souboru *Process.h*):

```
if (dynamic_cast<TMenuItem *>(Sender) == Pesunout1 )
  ChangeDlg->Caption = "Přesunout" ;
else if (dynamic_cast<TMenuItem *>(Sender) == Koprovat1)
  ChangeDlg->Caption = "Kopírovat";
else if (dynamic_cast<TMenuItem *>(Sender) == Pejmenovat1)
  ChangeDlg->Caption = "Přejmenovat";
else _c_exit();
ChangeDlg->CurrentDir->Caption = DirectoryOutline->Directory;
ChangeDlg->FromFileName->Text = FileList->FileName;
ChangeDlg->ToFileName->Text = "";
if((ChangeDlg->ShowModal()!=mrCancel)&&(ChangeDlg->ToFileName->Text!=""))
  ConfirmChange(ChangeDlg->Caption, ChangeDlg->FromFileName->Text,
                ChangeDlg->ToFileName->Text);
```

Aplikace nyní zobrazuje dialogové okno se správným titulkem a provádí požadovanou operaci. Aplikace někdy potřebuje provést jinou aplikaci. API Windows poskytuje funkci *ShellExecute*, která provádí aplikaci, ale tato funkce vyžaduje několik parametrů. Programová jednotka *FMXUtils* obsahuje její snadněji použitelnou alternativu, nazvanou *ExecuteFile*. *ExecuteFile* pracuje dvěma způsoby. Je-li ji předáno jméno proveditelného souboru, *ExecuteFile* spustí tuto aplikaci. Je-li ji předáno jméno dokumentu přiřazeného aplikaci, *ExecuteFile* spustí aplikaci a automaticky otevře tento dokument. *ExecuteFile* přebírá tři parametry typu *AnsiString* a čtvrtý parametr indikující způsob zobrazení okna aplikace. Tři řetězce reprezentují jméno souboru, parametry předané aplikaci a adresář použitý jako pracovní adresář aplikace. Poslední parametr může být jedna z konstant používaná API funkcí *ShowWindow*. Např. SW_SHOW zobrazí okno normálně, SW_SHOWMINIMIZED zobrazí okno minimalizované apod. Okno seznamu souborů obsahuje všechny informace potřebné k provedení souboru ze seznamu. Stejné příkazy můžeme připojit k události **OnDblClk** okna seznamu souborů, což umožní spouštět programy dvojitým kliknutím na jejich jménu v seznamu. Jestliže ale vybraný prvek je adresář, potom pravděpodobně chceme tento adresář otevřít (přejít na tento adresář). Naši obsluha události tedy bude vypadat takto:

```
if (HasAttr(FileList->FileName, faDirectory))
  DirectoryOutline->Directory = FileList->FileName;
else ExecuteFile(FileList->FileName," ",DirectoryOutline->Directory,SW_SHOW);
```

HasAttr je funkce typu bool z *FMXUtils*, která vrací *True*, když soubor uvedený jako její první parametr má atribut určený druhým parametrem; jinak vrací *False*. Nastavení vlastnosti **Directory** seznamu adresářů, způsobí změnu adresáře. Dvojitým kliknutím na adresáři v seznamu souborů lze nyní měnit adresář.

5. Přetažení prvku z formuláře může zjednodušit uživateli manipulaci s objekty formuláře. Můžeme přetáhnout celou komponentu nebo některý prvek z komponenty. Každý ovladač má vlastnost nazvanou **DragMode**, která určuje reakci komponenty, když uživatel zahájí její přetahování. Má-li **DragMode** hodnotu *dmAutomatic*, pak tažení začíná automaticky, když uživatel stiskne tlačítko myši s kurzorem myši na ovladači. Užitečnější je nastavit **DragMode** na *dmManual* (je to implicitní hodnota) a zahájit tažení v obsluze události stisknutí tlačítka myši. K zahájení manuálního tažení ovladače voláme metodu *BeginDrag* ovladače. *BeginDrag* přebírá parametr typu bool nazvaný *Immediate*. Jestliže předáme *True*, tažení začíná bezprostředně, stejně jako když je **DragMode** nastaveno na *dmAutomatic*. Jestliže předáme *False*, tažení začne až uživatel nepatrně přemístí myš. Volání *BeginDrag(False)* tedy umožňuje akceptovat kliknutí myší bez zahájení operace tažení. Můžeme také testovat, které tlačítko myši bylo stisknuto. Např. následující kód obsluhuje událost stisknutí tlačítka myši na okně seznamu souborů zahájením tažení a to pouze bylo-li stisknuto levé tlačítko myši:

```
void __fastcall TFMForm::FileListMouseDown(TObject *Sender,
                TMouseButton Button, TShiftState Shift, int X, int Y)
{
  if(Button==mbLeft) {
    if (dynamic_cast<TFileListBox *>(Sender)!=0) {
      if(dynamic_cast<TFileListBox *>(Sender)->ItemAtPos(Point(X,Y),
                                                         True) >=0)
        dynamic_cast<TFileListBox *>(Sender)->BeginDrag(False);
    }
  }
}
```

Když nyní spustíme naši aplikaci, vidíme, že můžeme přetahovat prvek z okna seznamu souborů, ale kurzor stále indikuje, že prvek nelze nikam umístit. Kurzor má kdekoliv nad obrazovkou stále tvar přeškrtnutého kruhu. Když uživatel táhne něco nad ovladačem, pak ovladač přijímá událost **OnDragOver**, ve které musíme indikovat, zda tažený prvek můžeme akceptovat, pokud by zde byl umístěn. Je-li prvek akceptovatelný, pak Builder změní tvar kurzoru. Pro akceptování prvku taženého nad ovladačem, vytvoříme obsluhu události **OnDragOver** ovladače. Tato událost má parametr volaný odkazem nazvaný *Accept*, který v případě akceptování prvku musíme nastavit na *True*. Nastavením *Accept* na *True* určujeme, že v případě uvolnění tlačítka myši v tomto místě, pro vložení taženého prvku, pak aplikace může zaslat událost **OnDragDrop** ovladači nad kterým je kurzor myši. Je-li *Accept* nastaveno na *False*, aplikace nemůže vložit prvek na ovladač a tedy tento ovladač nemusí mít obsluhu události **OnDragDrop**. Událost **OnDragOver** má několik parametrů, včetně zdroje tažení a současné pozice myši. Tyto parametry můžeme použít k určení, zda prvek akceptovat. Často ovladač akceptuje tažený prvek na základě typu zdroje tažení, ale může také akceptovat pouze prvek specifické instance. Následující kód způsobí akceptování taženého prvku adresářovým oknem pouze tehdy, je-li tažen z okna seznamu souborů:

```
void __fastcall TFMForm::DirectoryOutlineDragOver(TObject *Sender,
       TObject *Source, int X, int Y, TDragState State, bool &Accept)
{
```

```
  if (dynamic_cast<TFileListBox *> (Source)) Accept = True;
}
```

Když ovladač indikuje, že může akceptovat tažený prvek, musíme definovat způsob obsluhy vložení prvku. K zpracování vkládaného prvku, vytvoříme obsluhu události **OnDragDrop** pro ovladač akceptující prvek. Parametry **OnDragDrop** indikují zdroj tažení a souřadnice myši nad akceptujícím ovladačem. Tyto parametry předávají potřebné informace obsluze vkládání prvku. Např. adresářové okno akceptující prvek tažený z okna seznamu souborů může přemístit soubor z aktuální pozice do adresáře, kam je přetáhneme:

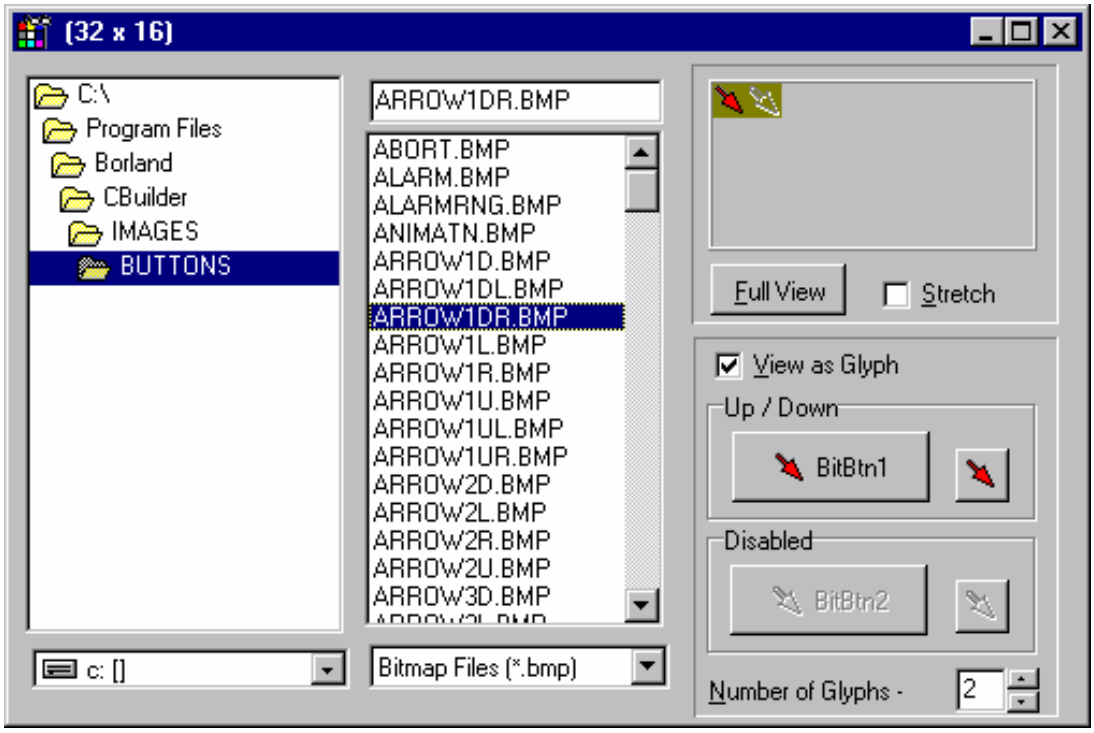
```
void __fastcall TFMForm::DirectoryOutlineDragDrop(TObject *Sender,
                         TObject *Source, int X, int Y)
{
  if (dynamic_cast<TFileListBox *> (Source)!=0)
    ConfirmChange("Přesunout", FileList->FileName,
    DirectoryOutline->GetItemPath(DirectoryOutline->ItemAtPos(Point(X,Y),
                                                        false)));
}
```

Přetažení má nyní stejný efekt jako použití **Soubor | Přesunout**, ale uživatel nemůže měnit jméno souboru. Když operace přetažení končí, a to buď vložením prvku nebo uvolněním tlačítka myši nad ovladačem neakceptujícím tažený prvek, Builder zasílá událost **OnEndDrag** zpět ovladači, ze kterého byl prvek tažen. Nejdůležitějším parametrem události **OnEndDrag** je parametr *Target* indikující, který ovladač tažený prvek převzal. Má-li *Target* hodnotu NULL, pak tažený prvek nebyl akceptován. V naši aplikaci ještě vytvoříme obsluhu události **OnEndDrag** okna seznamu souborů tak, aby byl aktualizován obsah aktuálního adresáře:

```
void __fastcall TFMForm::FileListEndDrag(TObject *Sender,
                                         TObject *Target, int X, int Y)
{
  if (Target != NULL) FileList->Update();
}
```

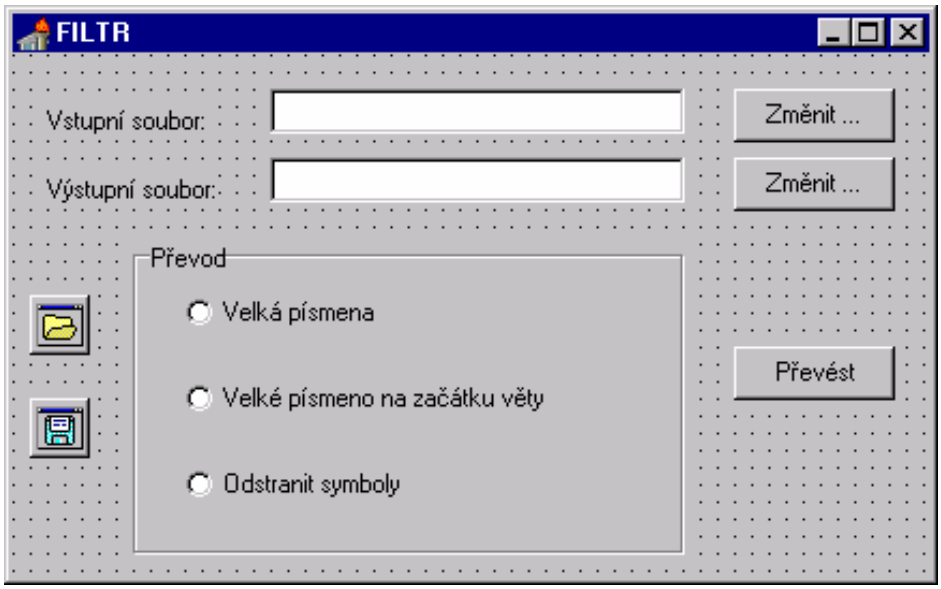
Tím je vývoj aplikace správce souborů ukončen.

6. Přidejte do předchozí aplikace nějakou další činnost (např. kopírování souborů přetažením myší při stisknuté klávese Ctrl).
7. Pokuste se vytvořit aplikaci pro prohlížení obrázků (viz následující obrázek). Komponentou **DriveComboBox** zvolíme diskovou jednotku, komponentou **DirectoryListBox** určíme adresář a v komponentě **FileListBox** vybereme zobrazovaný soubor. Soubory zobrazované v **FileListBox** určíme filtrem zadaným v komponentě **FilterComboBox**. Jméno zobrazovaného souboru uvedeme v editačním ovladači nad seznamem souborů. Tlačítkem **Full View** zobrazíme okno pro zobrazení obrázku v plné velikosti. Značkou **Stretch** určujeme, zda obrázek bude v normální velikosti nebo zda zaplní celou plánovanou plochu. Značkou **View as Glyph** určíme, zda ukázat, jak by bitová mapa byla zobrazena na demonstračních tlačítkách.



9. Různé aplikace

1. V další aplikaci se budeme zabývat prací se soubory. Z existujícího textového souboru se pokusíme vytvořit nový soubor s pozměněným obsahem. Program nazvaný FILTR umí převést všechny znaky textového souboru na velká písmena, zvětšit jen písmena každého prvního slova ve větě nebo odstranit znaky z druhé části tabulky ASCII (písmena s háčky a čárkami). Vytvoříme formulář podle následujícího obrázku (je zde použita komponenta **RadioGroup**):



Uživatel může zadat názvy vstupního a výstupního souboru ve dvou editačních ovladačích nebo stisknutím tlačítka **Změnit** otevřít příslušné dialogové okno pro výběr souboru. U komponenty **OpenDialog** nastavíme **Options** na *[ofPathMustExist, ofFileMustExist]* a u **SaveDialog** na *[ofOverwritePrompt, ofPathMustExist, ofCreatePrompt]*. Vlastnosti **Filter** u obou dialogových oken nastavíme na *Textový soubor (*.txt)*. Obsluha stisku horního tlačítka **Změnit** bude tvořena příkazem (pro spodní tlačítko ji vytvořte sami):

```
if (OpenDialog1->Execute()) Edit1->Text = OpenDialog1->FileName;
```

Do soukromé části deklarace formuláře vložíme tyto deklarace:

```
FILE *VstSoubor, *VystSoubor;
int DelkaSouboru;
void __fastcall PrevodVelka();
void __fastcall PrevodPrvniVelk();
```

```
void __fastcall PrevodSymb();
```
Obsluha stisku tlačítka **Převést** je tvořena příkazy (je zobrazeno dialogové okno, které bude popisovat proces převodu; tento další formulář bude obsahovat pouze komponentu **ProgressBar** a tlačítko **Ok**; tlačítko zakážeme):
```
if ((Edit1->Text != "")&&(Edit2->Text != "")) {
  if ((VstSoubor = fopen(Edit1->Text.c_str(), "rt")) == NULL){
    ShowMessage("Nelze otevřít vstupní soubor.");
    return;
  }
  fseek(VstSoubor, 0L, SEEK_END);
  DelkaSouboru = ftell(VstSoubor);
  fseek(VstSoubor, 0L, SEEK_SET);
  if ((VystSoubor = fopen(Edit2->Text.c_str(), "wt")) == NULL){
    ShowMessage("Nelze otevřít výstupní soubor.");
    return;
  }
  Form2->Show();
  Form2->BitBtn1->Enabled = false;
  Button3->Enabled = false;
  switch (RadioGroup1->ItemIndex) {
    case 0: PrevodVelka(); break;
    case 1: PrevodPrvniVelk(); break;
    case 2: PrevodSymb(); break;
  }
  fclose(VstSoubor);
  fclose(VystSoubor);
  Form2->BitBtn1->Enabled = true;
  Button3->Enabled = true;
}
else ShowMessage("Zadej jména souborù.");
```
Zbývá ještě vytvořit tři metody provádějící požadovaný převod. Následuje výpis těchto metod (je zde i pomocná funkce převádějící velká písmena na malá):
```
char __fastcall LowCase(char Zn)
{
  if ((Zn >= 'A') && (Zn <= 'Z')){return Zn - 'A' + 'a';}
  else return Zn;
}

void __fastcall TForm1::PrevodVelka()
{
  int Pozice = 0;
  while (!feof(VstSoubor)) {
    fputc(UpCase(fgetc(VstSoubor)), VystSoubor);
    Pozice++;
    Form2->ProgressBar1->Position = Pozice * 100 / DelkaSouboru;
    Application->ProcessMessages();
  }
}
void __fastcall TForm1::PrevodPrvniVelk()
{
  int Pozice = 0;
  char Zn;
  bool Tecka = true;
  while (!feof(VstSoubor)) {
    Zn = fgetc(VstSoubor);
    if ((Zn >= 'A') && (Zn <= 'Z')){
      if (Tecka) {fputc(Zn, VystSoubor); Tecka = false;}
      else {fputc(LowCase(Zn), VystSoubor); Tecka = false;}
    }
    else if ((Zn >= 'a') && (Zn <= 'z')){
      if (Tecka) {fputc(UpCase(Zn), VystSoubor); Tecka = false;}
      else {fputc(Zn, VystSoubor); Tecka = false;}
    }
    else if ((Zn == '.') || (Zn == '?') || (Zn == '!'))
```

```
      {fputc(Zn, VystSoubor); Tecka = true;}
    else {fputc(Zn, VystSoubor);}
    Pozice++;
    Form2->ProgressBar1->Position = Pozice * 100 / DelkaSouboru;
    Application->ProcessMessages();
  }
}
void __fastcall TForm1::PrevodSymb()
{
  int Pozice = 0;
  char Zn;
  while (!feof(VstSoubor)) {
    Zn = fgetc(VstSoubor);
    if (Zn < 128) fputc(Zn, VystSoubor);
    Pozice++;
    Form2->ProgressBar1->Position = Pozice * 100 / DelkaSouboru;
    Application->ProcessMessages();
  }
}
```

Tím je naše aplikace hotova a můžeme ji vyzkoušet. Pokuste se pochopit jak pracuje. Vidíme, že se soubory se pracuje stejným způsobem jako v C++.

2. Vedle používání textových souborů můžeme také běžným způsobem používat typové soubory. Nejprve vytvoříme aplikaci, která bude zobrazovat grafy (seznámíme se s komponentou **ChartFX**) a později k této aplikaci přidáme možnost ukládat a načítat zobrazené hodnoty do a ze souborů. Začneme s vývojem nové aplikace. Formulář zvětšíme a umístíme na ní komponentu **ChartFX** (bude zabírat asi horní dvě třetiny formuláře). Na zbývající plochu formuláře umístíme ještě komponentu **StringGrid** (s pěti sloupci a čtyřmi řádky, bez fixních řádků a sloupců, bez posuvníků a k vlastnosti **Options** přidáme hodnotu *goEditing*), komponentu **ComboBox** (vlastnost **Style** nastavíme na *csDropDownList* a prvky seznamu nastavíme na *1 – Lines, 2 – Bar, 3 – Spline, 4 – Mark, 5 – Pie, 6 – Area, 7 – Pareto, 8 – Scatter a 9 – HiLow*) a tlačítko s textem *Aktualizuj*. Mřížku řetězců můžeme editovat a aby přijímala pouze celá čísla musíme obsloužit její událost **OnGetEditMask**. Tuto obsluhu bude tvořit příkaz:

```
Value = "!09";
```

Tím máme možnost zadat do buněk mřížky jednomístné nebo dvojmístné číslo. Při vytváření formuláře vyplníme mřížku náhodnými hodnotami a voláním obsluhy stisku tlačítka *Aktualizuj* tyto hodnoty zobrazíme v grafu. Do veřejné části deklarace formuláře přidáme:

```
bool Modifikovano;
AnsiString AktualniSoubor;
```

Obsluha **OnCreate** formuláře bude tedy tvořena příkazy:

```
Randomize();
for (int I = 0; I < 5; I++)
  for (int J = 0; J < 4; J++)
    StringGrid1->Cells[I][J] = IntToStr(random(100));
Button1Click(this);
ComboBox1->ItemIndex = ChartFX1->ChartType - 1;
Modifikovano = true;
AktualniSoubor = "";
```

a obsluhu stisku tlačítka *Aktualizuj* tvoří příkazy (nápověda k používání komponenty **ChartFX** je umístěna v adresáři CBuilder\OCX):

```
ChartFX1->OpenDataEx(COD_VALUES,5,4);
for (int I = 0; I < 5; I++){
  ChartFX1->ThisSerie = I;
  for (int J = 0; J < 4; J++)
    ChartFX1->Value[J] = StrToIntDef(StringGrid1->Cells[I][J], 0);
}
ChartFX1->CloseData(COD_VALUES);
Modifikovano = true;
```

Zbývá ještě vytvořit obsluhu události **OnChange** kombinovaného ovladače. Tvoří ji příkazy:

```
ChartFX1->ChartType = ComboBox1->ItemIndex + 1;
Modifikovano = true;
```

Tím je tato část aplikace hotova. Vyzkoušejte jak pracuje a pokuste se pochopit, jak lze ovládat komponentu **ChartFX**.

3. Aplikaci vytvořenou v předchozím zadání rozšíříme o možnost ukládání zobrazovaných dat do souboru. K aplikaci přidáme nabídku **Soubor** s volbami *Otevřít*, *Uložit* a *Uložit jako* (můžete přidat i volbu *Konec*) a přidáme komponenty **OpenDialog** a **SaveDialog** (nastavím vhodně jejich vlastnosti). Obsluha volby *Otevřit* bude tvořena příkazy:

```
FILE *LoadFile;
if (OpenDialog1->Execute()) {
  AktualniSoubor = OpenDialog1->FileName;
  Caption = "Graf " + AktualniSoubor;
  if ((LoadFile = fopen(AktualniSoubor.c_str(), "rb")) == NULL){
    ShowMessage("Nelze otevřít soubor");
    return;
  }
  int Hodnota;
  for (int I = 0; I < 5; I++)
    for (int J = 0; J < 4; J++){
      fread(&Hodnota, sizeof(Hodnota), 1, LoadFile);
      StringGrid1->Cells[I][J] = IntToStr(Hodnota);
    }
  fread(&Hodnota, sizeof(Hodnota), 1, LoadFile);
  ComboBox1->ItemIndex = Hodnota;
  Modifikovano = false;
  fclose(LoadFile);
  ComboBox1Change(this);
  Button1Click(this);
}
```

   Zbývající obsluhy vytvořte sami a aplikaci vyzkoušejte.

4. Další zajímavou oblastí Builderu je podpora souborových proudů. Knihovna VCL obsahuje abstraktní třídu **TStream** a její tři potomky: **TFileStream**, **THandleStream** a **TMemoryStream**. Pro nahrávání dat ze souboru a ukládání do souboru můžeme použít dva souborově orientované proudy. **THandleStream** se používá v případě, kdy již máme madlo souboru. **TFileStream** použijeme, když máme jen jméno souboru. Třetí proudovou třídou je **TMemoryStream**, která pracuje s pamětí a nikoli se skutečným souborem. Tato třída však obsahuje speciální metody pro kopírování svého obsahu do nebo z jiného proudu, který může být souborovým proudem. Proudy mohou nahradit tradiční soubory. Jejich velkou výhodou je např. to, že můžeme pracovat s paměťovými proudy a potom je uložit do souboru. Tímto způsobem lze zvýšit rychlost programu intenzivně pracujícího se soubory.

   Zvláště důležitou vlastností proudů je jejich schopnost přenášet komponenty. Všechny třídy knihovny VCL jsou potomci **TPersistent**, speciální třídy používané k ukládání objektů do proudů a obsahují metody pro ukládání a nahrávání všech vlastností a veřejných položek. Proto mohou všichni potomci třídy **TComponent** ukládat sami sebe do proudu nebo mohou být nahráním z proudu automaticky vytvořeni. Program k tomu může využívat metod proudu *WriteComponent* a *ReadComponent*. Proudy v zásadě nevědí nic o čtení nebo zápisu komponent. Metody tříd **TStream** jednoduše používají dvě jiné třídy:**TReader** a **TWriter**, obě potomky třídy **TFiler**. Objekty **TReader** a **TWriter** používají proud, na který se vztahují a jsou schopné mu přidělit speciální značky pro provedení kontroly datového formátu. Objekt **TWriter** může uložit značku komponenty, potom uložit komponentu, všechny její vlastnosti a všechny komponenty, které obsahuje. Obsahuje metodu *WriteRootComponent*, která uloží komponentu předanou jako parametr a také všechny komponenty, které obsahuje. Podobně třída **TReader** obsahuje metodu *ReadRootComponent*, která je schopná vytvořit nové objekty s využitím informace o třídách uložených v proudu. To je možné pod jednou podmínkou: název komponenty musí být aplikací registrován (funkcí *RegisterClasses*).

   Vraťme se ale ke konkrétní aplikaci. Začneme vývojem nové aplikace. Formulář této aplikace je jednoduchý. Na horní okraj formuláře vložíme komponentu **Panel** (zrušíme jeho titulek) a na ní vložíme tři voliče s texty *Label*, *Edit* a *Button*. (první z nich nastavíme). Při kliknutí myší na formuláři, vytvoříme komponentu určenou vybraným voličem. Před deklaraci typu formuláře vložíme deklaraci výčtového typu

```
enum Typ {Label, Edit, ButtonX};
```

   popisující vybraný volič a do soukromé části deklarace formuláře umístíme:

```
int Citac;
Typ Odkaz;
```

   Vytvořte obsluhy kliknutí na voličích tak, aby v položce *Odkaz* byla uložena informace o vybraném voliči. Dále vytvoříme obsluhu **OnCreate** formuláře s těmito příkazy:

```
Odkaz = Label;
Citac = 0;
```

Po vytvoření obsluhy **OnMouseDown** formuláře již můžeme aplikaci vyzkoušet. Tuto obsluhu tvoří příkazy:

```
TControl *Objekt;
AnsiString Nazev;
switch (Odkaz) {
  case Label: Objekt = new TLabel(this); break;
  case Edit: Objekt = new TEdit(this); break;
  case ButtonX: Objekt = new TButton(this); break;
}
Objekt->Parent = this;
Objekt->Left = X;
Objekt->Top = Y;
Citac++;
Nazev = String(Objekt->ClassName()) + IntToStr(Citac);
Nazev.Delete(1, 1);
Objekt->Name = Nazev;
Objekt->Visible = true;
```

Klikáním myši na formuláři vytváříme komponenty určené vybraným voličem. Vyzkoušejte.

5. Této aplikaci dále umožníme ukládání vložených komponent do souboru a jejich opětovnému nahrání ze souboru. K aplikaci přidáme nabídku **Soubor** s volbami: *Zrušit*, *Otevřít*, *Uložit jako* a *Konec*. Přidáme také komponenty **OpenDialog** a **SaveDialog** (kde nastavíme vhodné vlastnosti; u našich souborů budeme používat příponu CMP). Tyto komponenty musíme vložit na již použitou komponentu **Panel**. Obsluhu volby Zrušit tvoří příkazy (zrušíme všechny komponenty mimo komponent vložených na panel):

```
for (int I = ControlCount-1; I >= 0; I--)
  if (String(Controls[I]->ClassName()) != "TPanel") Controls[I]->Free();
Citac = 0;
```

Komponenty ukládané do proudu musíme registrovat a tedy do obsluhy **OnCreate** formuláře přidáme příkazy:

```
TComponentClass classes[3] =
             {__classid(TEdit), __classid(TLabel), __classid(TButton)};
RegisterClasses(classes, 2);
```

Vytvoříme ještě obsluhu volby *Uložit jako*. Tvoří ji příkazy:

```
if (SaveDialog1->Execute()){
  TFileStream *S;
  S = new TFileStream(SaveDialog1->FileName, fmOpenWrite | fmCreate);
  for (int I = 0; I < ControlCount; I++)
    if (String(Controls[I]->ClassName()) != "TPanel")
      S->WriteComponent(Controls[I]);
  delete S;
}
```

Obsluhu volby *Otevřít* tvoří:

```
if (OpenDialog1->Execute()) {
  TFileStream *S;
  TComponent *Novy;
  Zruit1Click(this);
  S = new TFileStream(OpenDialog1->FileName, fmOpenRead);
  while (S->Position < S->Size){
    Novy = S->ReadComponent(NULL);
    InsertControl(dynamic_cast<TControl *> (Novy));
    Citac++;
  }
  delete S;
}
```

Zbývající obsluhy vytvořte sami. Aplikaci vyzkoušejte.

6. V další aplikaci se seznámíme s používáním funkce *MessageBeep*. Začneme s vývojem nové aplikace a na formulář umístíme skupinu pěti voličů s texty *mb_IconAsterisk*, *mb_IconExclamation*, *mb_IconHand*, *mb_IconQuestion* a *mb_Ok* (jsou to jména standardních zvuků Windows) a tři tlačítka s texty *Test*, *Beep 0xFFFF* a *Beep Sound*. Pokud chceme zjistit, zda v našem počítači je instalován ovladač zvuku (např. máme zvukovou kartu), pak stiskneme tlačítko *Test*. Toto zjištění je provedeno funkcí *waveOutGetNumDevs* (funkce je v hlavičkovém souboru *Mmsystem.h*). Obsluha stisku tohoto tlačítka je tvořena těmito příkazy:

```
if (waveOutGetNumDevs() > 0)
  MessageDlg("Zvuky jsou podporovány", mtInformation,
             TMsgDlgButtons() << mbOK, 0);
```

```
else
  MessageDlg("Zvuky nejsou podporovány", mtInformation,
             TMsgDlgButtons() << mbOK, 0);
```

Tlačítko *Beep 0xFFFF* vydává standardní zvuk. Jeho obsluhu tvoří příkaz:

```
MessageBeep(0xFFFF);
```

Tlačítko *Beep Sound* přehraje zvuk podle vybraného voliče. Jeho obsluha je tvořena příkazy:

```
unsigned short Priznak;
switch (RadioGroup1->ItemIndex) {
  case 0: Priznak = MB_ICONASTERISK; break;
  case 1: Priznak = MB_ICONEXCLAMATION; break;
  case 2: Priznak = MB_ICONHAND; break;
  case 3: Priznak = MB_ICONQUESTION; break;
  case 4: Priznak = MB_OK; break;
}
MessageBeep(Priznak);
```

Tím jsme dokončili vývoj této aplikace. Pokud máte v počítači zvukovou kartu, pak ji můžete vyzkoušet.

7. Funkce *sndPlaySound* (je definovaná v **Mmsystem.h**) může být použita k přehrávání systémových zvuků a souborů WAV. Na formulář umístíme komponentu **ListBox**, ve které uvedeme některé systémové zvuky a soubory WAV (např. *SystemStart*, *SystemExit*, *SystemQuestion*, *Bell.wav* a *Dog.wav*) a tlačítka *Hraj*, *Zastav* a *Cyklus*. V obsluze **OnCreate** formuláře vynulujeme vlastnost **ItemIndex** komponenty **ListBox**. Obsluha stisku tlačítka *Hraj* je tvořena příkazy:

   ```
   char XX[100];
   sndPlaySound(strcpy(XX,
     ListBox1->Items->Strings[ListBox1->ItemIndex].c_str()), SND_ASYNC);
   ```

   Tento kód může být použit k přehrávání jak systémových zvuků, tak souborů WAV. Pokud položka seznamu nemá žádný odpovídající soubor nebo systémový zvuk, pak se zahraje implicitní zvuk. Druhý parametr udává, že má funkce přehrát zvuk asynchronně a ihned vrátit řízení systému. Pokud zde použijeme *SND_SYNC* pak funkce nevrátí řízení, pokud není celý zvuk kompletně přehrán. V našem případě (při asynchronním přehrávání) můžeme ale zvuk zastavit opětovným voláním stejné funkce, bez prvního parametru (obsluha stisku tlačítka *Zastav*):

   ```
   sndPlaySound(NULL, 0);
   ```

   Poslední tlačítko provádí opakované přehrávání zvuku. Jeho obsluha je tvořena příkazy:

   ```
   char XX[100];
   sndPlaySound(strcpy(XX,
     ListBox1->Items->Strings[ListBox1->ItemIndex].c_str()),
     SND_ASYNC | SND_LOOP);
   ```

   Aplikace je hotova a pokud vlastníme zvukovou kartu, pak si můžeme vyzkoušet přehrávání různých souborů WAV.

8. Komponenta **MediaPlayer** obsahuje většinu schopností Media Control Interface (MCI) Windows, vysokoúrovňového rozhraní pro řízení vnitřních a vnějších multimediálních zařízení. Především si povšimněme vlastnosti **DeviceType**. Může nabývat hodnoty *dtAutoSelect*, která značí, že druh zařízení závisí na příponě aktuálního souboru. Alternativně můžeme zadat např. *dtAVIVideo*, *dtCDAudio*, *dtWaveAudio* nebo mnoho jiných. Je to jediný přístup, který můžeme použít pro zařízení netýkající se souboru, jako čtecí zařízení zvukového CD. Jestliže je druh zařízení i soubor vybrán, můžeme dané zařízení otevřít (nebo nastavit hodnotu vlastnosti **AutoOpen** na *True*) a tlačítka komponenty **MediaPlayer** budou přístupná (ne všechny jsou smysluplná pro každý druh média). Komponenta má tři vlastnosti týkající se tlačítek: **VisibleButtons**, **EnabledButtons** a **ColoredButtons**. První určuje, která tlačítka jsou přítomná v ovladači, druhá povolená tlačítka a třetí tlačítka s barevným označením. Komponenta má také událost **OnClick**. Tato událost je ale jiná než obvykle, protože obsahuje parametr udávající, které tlačítko bylo stisknuto a další parametr můžeme použít k potlačení implicitní akce spojené s tlačítkem. Událost **OnNotify** informuje komponentu o úspěšnosti akce generované tlačítkem. Jiná událost **OnPostClick** je generována buď jakmile akce začne nebo když skončí (v závislosti na hodnotě vlastnosti **Wait**; určuje zda operace mají či nemají být synchronní).

   Začneme s vývojem nové aplikace. K hornímu okraji formuláře vložíme komponentu **Label** s textem *Soubor:* a vedle další komponentu **Label** jejíž text zrušíme a nazveme ji *FileLabel*. Pod tyto komponenty vložíme tlačítko s textem *Nový soubor ...*. Obsluhu stisku tohoto tlačítka tvoří příkazy (na formulář musíme vložit i **OpenDialog**, kde nastavíme vlastnost **Filter** na *Soubory wav (*.wav)* a *Soubory midi (*.mid)*):

   ```
   if (OpenDialog1->Execute()) {
     FileLabel->Caption = OpenDialog1->FileName;
     MediaPlayer1->FileName = OpenDialog1->FileName;
     MediaPlayer1->Open();
   ```

```
  MediaPlayer1->Notify = True;
}
```

Pod tlačítkem umístíme nad sebe dvě komponenty **Label** s texty *Zpráva:* a *Akce:* a vedle ně další dvě komponenty **Label** jejich text opět zrušíme a nazveme je *NotifLabel* a *ActionLabel*. Pod tyto komponenty vložíme komponentu **MediaPlayer**, kde necháme zobrazit pouze prvních pět tlačítek. Pro tuto komponentu vytvoříme obsluhy událostí **OnClick** a **OnNotify**. První obsluhu tvoří příkazy:

```
switch (Button) {
  case btPlay: ActionLabel->Caption = "Playing"; break;
  case btPause: ActionLabel->Caption = "Paused"; break;
  case btStop: ActionLabel->Caption = "Stopped"; break;
  case btNext: ActionLabel->Caption = "Next"; break;
  case btPrev: ActionLabel->Caption = "Previous"; break;
}
```

a druhou příkazy:

```
switch (MediaPlayer1->NotifyValue){
  case nvSuccessful: NotifLabel->Caption = "Success"; break;
  case nvSuperseded: NotifLabel->Caption = "Superseded"; break;
  case nvAborted: NotifLabel->Caption = "Aborted"; break;
  case nvFailure: NotifLabel->Caption = "Failure"; break;
}
MediaPlayer1->Notify = true;
```

Aplikace je hotova a pokud vlastníme zvukovou kartu, pak si můžeme přehrát nějaký zvukový soubor.

9. V další aplikaci se budeme zabývat videem. Tuto aplikaci můžeme spustit pouze, máme-li na našem počítači některý z ovladačů pro video (např. Microsoft Video for Windows). Začneme s vývojem nové aplikace, na formulář umístíme komponentu **MediaPlayer**, do její vlastnosti **FileName** zadáme některý soubor AVI a vlastnost **AutoOpen** nastavíme na *True*. Nyní již aplikaci můžeme spustit a stiskem tlačítka **Play** spustíme přehrávání zadaného videa.
10. Jestliže nechceme video spouštět ve zvláštním okně, pak na formulář můžeme vložit např. komponentu **Panel** a vlastnosti **Display** komponenty **MediaPlayer** určíme, kde se video bude zobrazovat. Zde mohou ale vzniknout problémy s rozměry zobrazovací plochy. Vyzkoušejte.
11. Pokuste se vytvořit aplikaci pro přehrávání zvukových CD.

10. Dynamická výměna dat DDE

1. DDE umožňuje dvěma aplikacím zřídit spojení a používat jej pro přenos dat (komunikovat formou rozhovoru). Jedna z aplikací účastnící se rozhovoru je označována jako server a druhá jako klient. Server vlastně informace poskytuje; klient je aplikace, která proces řídí. Každá aplikace může působit jako server pro více klientů nebo jako klient pro více serverů a také současně jako klient i server.

   Úloha serveru je v podstatě posílat data klientovi. Role klienta spočívá v zahájení rozhovoru, požádání serveru o data nebo požádat server o provedení příkazů (*execute*). Jakmile je rozhovor aktivní, může klient požádat server o data (*request*) nebo spustit *avizovaný cyklus*. To znamená, že klient může požádat server, aby jej informoval o každé změně určité části dat. Server doručí buď oznámení nebo novou kopii dat při každé jejich změně. Pro určení serveru, na který se můžeme připojit a předmětu výměny používá DDE tři prvky: **Service** je v zásadě název aplikace DDE serveru. Může to být název spustitelného souboru (bez přípony), avšak může jít i jiný název určený přímo serverem. **Topic** je globální téma rozhovoru. Může to být datový soubor, okno serveru nebo cokoliv jiného. DDE rozhovor je mezi klientem a serverem navázán o určitém tématu. **Item** je identifikátor určitého datového prvku. Může se jednat o položku databáze, buňku tabulky. Pomocí jednoduchého rozhovoru si mohou klient a server vyměňovat údaje o mnoha položkách.

   Nejprve se pokusíme vytvořit pravděpodobně nejjednodušší možný server, který bude připojen k nejjednoduššímu možnému klientovi. Server našeho příkladu (aplikaci nazveme PRVSERV) obsahuje editační komponentu a komponentu **DdeServerItem**. Pro editační komponentu vytvoříme obsluhu události **OnChange** je tvořen tímto příkazem (při každé změně obsahu editačního ovladače je jeho obsah překopírován do vlastnosti **Text** komponenty **DdeServerItem**):

   ```
   DdeServerItem1->Text = Edit1->Text;
   ```

   Tím je aplikace serveru hotova. Aplikace klienta je jej nepatrně složitější. Formulář zde obsahuje komponenty **DdeClientConv**, **DdeClientItem**, editační ovladač a tlačítko (s textem *Připojit*). Vlastnost **DdeConv** u **DdeClientItem** nastavíme na *DdeClientConv1*. V době návrhu neexistuje žádné spojení mezi **DdeClientConv** a serverem. Spojení je inicializováno po stisku tlačítka (obsluha jeho stisku je tvořena příkazy):

```
if (DdeClientConv1->SetLink("PRVSERV", "Form1")) {
  ShowMessage("Spojeno");
  DdeClientItem1->DdeItem = "DdeServerItem1";
}
else ShowMessage("Chyba");
```

Je zde volána metoda *SetLink*, která jako parametry předá službu (název serveru) a téma (hlavičku formuláře). Pokud je spojení úspěšné, nastaví aplikace vlastnost **DdeItem** prvku klienta na prvek serveru. Když je spojení navázáno a vazba je aktivní, začne server aplikace posílat klientovi data při každé změně dat na serveru. Vytvoříme ještě obsluhu **OnChange** komponenty **DdeClientItem**. Je tvořena příkazem:

```
Edit1->Text = DdeClientItem1->Text;
```

Aplikace klienta je nyní také hotova. Pokud spustíme aplikaci serveru i aplikaci klienta a stiskneme tlačítko *Připojit* je navázáno spojeni. Vše co nyní zapíšeme do editačního ovladače v aplikaci serveru je zobrazeno v editačním ovladači aplikace klienta. Vyzkoušejte. Obě editační komponenty jsou nyní propojeny pomocí DDE.

2. V další aplikaci budeme demonstrovat podporu DDE pro kopírování a vlepování. Prvním krokem bude vytvoření nové verze serveru DDE. Formulář nového serveru obsahuje tři editační ovladače spojené s komponentami **DdeServerItem** a tři tlačítka (texty *Kopíruj*). Formulář obsahuje také komponentu **DdeServerConv**. Do editačních ovladačů vložíme texty: *První řádek*, *Druhý řádek* a *Třetí řádek*. U komponent **DdeServerItem** nastavíme vlastnost **ServerConv** na **DdeServerConv1**. V obsluze události **OnCreate** formuláře jsou příkazy:

   ```
   DdeServerItem1->Text = Edit1->Text;
   DdeServerItem2->Text = Edit2->Text;
   DdeServerItem3->Text = Edit3->Text;
   ```

   Obsah editačních komponent je při vytváření formuláře zkopírován do odpovídajících prvků serveru. Při změně některého editačního ovladače text musíme opět zkopírovat. Vytvoříme tedy obsluhy událostí **OnChange** editačních ovladačů. Pro **Edit1** obsluhu bude tvořit příkaz (ostatní vytvořte sami):

   ```
   DdeServerItem1->Text = Edit1->Text;
   ```

   Je nutno vytvořit i obsluhy stisku tlačítek. Tyto obsluhy jsou opět podobné. Pro tlačítko spojené s **Edit1** jsou zde příkazy:

   ```
   Clipboard()->Open();
   Clipboard()->AsText = DdeServerItem1->Text;
   DdeServerItem1->CopyToClipboard();
   Clipboard()->Close();
   ```

   Operace kopírování prvku okopíruje informaci o vazbě. Tato informace nejsou vlastní data, ale spíše text. Proto jsou potřebné obě kopie. Pokud vytvoříme pouze vazbu, potom mnoho aplikací taková data nepozná, protože jsme nespecifikovali žádné informace ohledně formátu. S tímto programem můžeme kopírovat text jednoho z editačních ovladačů této aplikace a nalepit jej takřka do libovolného klienta DDE. Např. v MS Wordu k nalepení slouží příkaz **Úpravy | Vložit jinak**. Spustíme náš program, zkopírujeme tlačítkem jeden editační ovladač, otevřeme dokument Wordu a zvolíme **Úpravy | Vložit jinak**. V zobrazeném dialogovém okně vybereme **Vložit propojení** a stiskneme **Ok**. Dojde k vložení textu, ale přepneme-li se do našeho serveru a změníme text, pak se tento text změní i v dokumentu Wordu.

3. Nyní, když vidíme, že náš server funguje, můžeme vytvořit nového klienta schopného nalepovat data. Pokud máme pouze jeden zdroj dat, můžeme zřídit vazbu v době návrhu. Jestliže server DDE běží, potom v klientské aplikaci DDE při návrhu můžeme zřídit spojení přes dialogové okno DDE Info. Vybereme komponentu **DdeClientConv** a otevřeme editor vlastnosti **DdeService**. Zde můžeme nadefinovat požadované spojení (zadáme **Service** a **Topic** serveru) a stiskneme tlačítko **Paste Link**, čímž navážeme spojení se serverem. Formulář naší druhé klientské aplikace DDE se podobá formuláři první klientské aplikace. Namísto tlačítka *Připojit* jsou zde dvě tlačítka s texty *Paste* a *Paste Link*. U komponenty **DdeClientItem** nastavíme vlastnost **DdeConv** na *DdeClientConv1*. Editační komponenta je opět spojena s prvkem klienta. Obsluha **OnChange** komponenty **DdeClientItem** je tedy tvořena příkazem:

   ```
   Edit1->Text = DdeClientItem1->Text;
   ```

   Tlačítko *Paste* prostě zkopíruje text schránky a zruší spojení DDE:

   ```
   if (Clipboard()->HasFormat(CF_TEXT)) {
     Edit1->Text = Clipboard()->AsText;
     DdeClientConv1->CloseLink();
   }
   else MessageBeep(0xFFFF);
   ```

   Obsluha stisku druhého tlačítka obsahuje příkazy:

   ```
   AnsiString Service, Topic, Item;
   ```

```
if (Clipboard()->HasFormat(ddeMgr->LinkClipFmt)){
  GetPasteLinkInfo(Service, Topic, Item);
  DdeClientConv1->SetLink(Service, Topic);
  DdeClientItem1->DdeItem = Item;
}
else MessageBeep(0xFFFF);
```

Nastavením těchto hodnot je spojení automaticky navázáno, jelikož implicitní hodnota vlastnosti **ConnectMode** klientovy komponenty **DdeClientConv** je nastavena na *ddeAutomatic*. Když spustíme jak server, tak i klientskou aplikaci, pak můžeme provádět kopie textu z jednoho ze tří editačních ovladačů nebo zřídit vazbu.

4. Další typické použití DDE je předávání příkazů jinému programu. Správce programu umožňuje vytvářet nové programové skupiny. Pro komunikaci se správcem programů potřebujeme do formuláře umístit konverzační komponentu klienta (**DdeClientConv**), vlastnosti **DdeService** a **DdeTopic** nastavit na *PROGMAN* a pro vytvoření nové skupiny provést příkaz (můžeme jej např. umístit do obsluhy stisku tlačítka):

   ```
   DdeClientConv1->ExecuteMacro("[CreateGroup(""Nova skupina"")]", false);
   ```

   První parametr je makro, které chceme provést. Vyzkoušejte.

5. Dále vytvoříme server DDE automaticky měnící data. Aplikaci nazveme DATASERV. Aplikace má jednoduchý formulář. Obsahuje komponentu **Memo** s vlastností **ReadOnly** nastavenou na *True* (vložíme do ní pět řádků s hodnotami 40, 50, 60, 70 a 80 a komponentu zakážeme). Formulář dále obsahuje komponenty **Timer**, **DdeServerItem** a tlačítko s textem *Kopíruj*. Obsluha časovače bude tvořena příkazy:

   ```
   for (int I = 0; I < 5; I++) {
     int Hodnota = StrToIntDef(Memo1->Lines->Strings[I], 50);
     Hodnota = Hodnota + random(21) - 10;
     Memo1->Lines->Strings[I] = IntToStr(Hodnota);
   }
   DdeServerItem1->Lines = Memo1->Lines;
   ```

   Nyní po spuštění aplikace se hodnoty v komponentě **Memo** automaticky mění. Obsluha stisknutí tlačítka je tvořena příkazy (do schránky překopírujeme data i vazbu):

   ```
   Clipboard()->Open();
   Clipboard()->AsText = Memo1->Text;
   DdeServerItem1->CopyToClipboard();
   Clipboard()->Close();
   ```

   Tím je aplikace serveru hotova.

6. Klientská aplikace obsahuje pouze komponenty **DdeClientConv** a **DdeClientItem** (propojíme je). Obsluhu události **OnCreate** formuláře tvoří příkazy (při vytvoření formuláře je zřízeno spojení):

   ```
   if (DdeClientConv1->SetLink("DATASERV", "Form1"))
     DdeClientItem1->DdeItem = "DdeServerItem1";
   else ShowMessage("Spusť server před klientem!");
   ```

   Jako soukromou položku formuláře vložíme (pro uložení hodnot serveru):

   ```
   int Hodnoty[5];
   ```

   Aby při přenosu dat ze serveru (přenášíme řetězec znaků) se nepřeskakovali znaky odřádkování, musíme nastavit vlastnost **FormatChars** u **DdeClientConv** na *True*. Obsluha **OnChange** komponenty **DdeClientItem** je tvořena příkazy (data ze serveru uložíme do našeho pole):

   ```
   for (int I = 0; I < DdeClientItem1->Lines->Count; I++)
     Hodnoty[I] = StrToIntDef(DdeClientItem1->Lines->Strings[I], 50);
   Invalidate();
   ```

   Vlastní zobrazování dat budeme provádět v obsluze **OnPaint** formuláře:

   ```
   int DX = ClientWidth / 11;
   int DY = ClientHeight / 3;
   float Meritko = DY / 100.0;
   Canvas->Pen->Width = 3;
   Canvas->MoveTo(0, DY * 2);
   Canvas->LineTo(ClientWidth, DY * 2);
   Canvas->Pen->Width = 1;
   Canvas->MoveTo(0, DY);
   Canvas->LineTo(ClientWidth, DY);
   for (int I = 0; I < 5; I++){
     if (Hodnoty[I] > 0) Canvas->Brush->Color = clGreen;
     else Canvas->Brush->Color = clRed;
     Canvas->Rectangle(DX * (2*I+1), DY*2-floor(Hodnoty[I]*Meritko),
                       DX * (2*I+2), DY*2);
   ```

```
}
```

Tím je klientská aplikace hotova. Můžeme vyzkoušet zobrazování dat získaných ze serveru. Vidíme, že pomocí DDE můžeme získávat různá data (většinou je vhodné převést je na text).

11. Automatizace OLE

1. V této kapitole budeme používat objekty Builderu k vytvoření serveru a klienta automatizace OLE. Vytvoření serverů a klientů automatizace OLE není obtížné. Nicméně je zde několik komplikujících situací. Automatizace OLE umožňuje přistupovat k objektům, které nejsou umístěny pouze v našem programu, ale v jiných programech našeho systému. Přesněji řečeno, můžeme přistupovat k metodám a vlastnostem těchto objektů, ale ne k jejich datům. Tento přístup lze provádět bez ohledu na programovací jazyk použitý k implementování objektu. Jsou dva hlavní typy automatizace OLE: servery OLE a kontejnery (nebo klienti) OLE. Servery automatizace poskytují funkčnost, která může být zpřístupněna kontejnerům automatizace. V tomto smyslu, aplikace nebo DLL, která hostí objekt je nazvaná server, a aplikace nebo DLL, která k němu přistupuje je nazvaná kontejner. Builder dává flexibilitu k integraci aplikací a DLL s různými aplikacemi serverů nebo kontejnerů OLE. Klasickými příklady serverů automatizace OLE jsou MS Word a Excel. Obě tyto aplikace mohou být řízeny aplikací Builderu, nebo jiným kontejnerem.

   K vytvoření jednoduché aplikace kontejneru OLE (klienta OLE) umístíme na formulář komponentu **OleContainer**, zvětšíme ji na celou plochu formuláře, odstraníme její rámeček (vlastnost **BorderStyle**), klikneme na ní pravým tlačítkem myši a v místní nabídce zvolíme **Insert Object**. Objeví se před námi dialogové okno, které nám umožňuje zvolit aplikační server. Seznam serverů, které se objeví v seznamu závisí na aplikacích OLE instalovaných na našem systému. Vybereme požadovaný server (např. Bitmap Image) a stiskneme **OK**. Tím je vývoj naší aplikace hotov. Aplikaci spustíme a dvojitým kliknutím na kontejneru OLE spustíme zadaný server OLE v našem formuláři. To je dáno implicitní hodnotou vlastnosti **AutoActive** (*aaDoubleClick*).

2. Začneme vývojem nové aplikace. Formulář bude obsahovat **OleContainer**, nabídku a panel. V této verzi neobsahuje kontejner předdefinovaný OLE objekt. Nový OLE objekt bude vytvořen příkazem **Soubor | Nový**. Nabídka bude obsahovat příkazy (v závorkách jsou uvedeny hodnoty **GroupIndex**):

```
Soubor (0)          Editace (1)          Nápověda (5)
Nový                Vyjmout              O aplikaci
Otevřít ...         Kopírovat
Uložit jako ...     Vložit
--------------      Vložit jinak ...
Konec               ----------------
                    Propojení
                    Objekt
```

   Většinu voleb v nabídce zakážeme (až na *Nový*, *Konec* a *O aplikaci*). U komponenty **OleContainer** nastavíme **Align** na *alClient*, **BorderStyle** na *bsNone* a **Ctrl3D** na *False*. U panelu nastavíme vlastnosti **Align** na *alBottom*, **Locked** na *True* a na panel vložíme tlačítko s textem *Vlastnosti ...* . Další důležitou položkou nabídky je **Editace | Objekt**. Je spojena s OLE kontejnerem. Toto spojení vytvoříme nastavením vlastnosti **ObjectMenuItem** formuláře na hodnotu *Objekt1*. Tato volba nabídky je automaticky přístupná, když je zvolen OLE kontejner (není ale aktivní) a obsahuje uvnitř objekt. V tomto případě se text položky nabídky změní, aby odrážel OLE objekt a podnabídka tohoto prvku bude obsahovat seznam akcí, které lze provádět s objektem. Aplikace s OLE kontejnerem také podporují slučování nabídek.

   Jediná věc, kterou musíme naprogramovat je obsluha volby **Nový** v nabídce. Tuto obsluhu tvoří příkaz:

```
OleContainer1->InsertObjectDialog();
```

   Povšimněte si jednoduchého nástrojového pruhu umístěného v dolní části formuláře a také tlačítka **Vlastnosti**, které je jediným tlačítkem tohoto nástrojového pruhu. Přítomnost nástrojového pruhu umožňuje deaktivovat komponentu **OleContainer**, aktivovat ji bez editace objektu apod. Stačí kliknout na tlačítko a na kontejner pro změnu zaostření mezi těmito dvěma komponentami. Aby nástrojový pruh zůstal viditelný, zatímco dochází k editaci, měl by mít panel nastavenou vlastnost **Locked** na *True*. To umožní panelu zůstat trvale přítomným v aplikaci a zabránění jeho nahrazení nástrojovým pruhem serveru.

   Tlačítko **Vlastnosti** zobrazí dialogové okno (list vlastností OLE objektu). Obsluha stisku tohoto tlačítka je tvořena příkazem:

```
OleContainer1->ObjectPropertiesDialog();
```

   Aplikace je hotova a můžeme ji vyzkoušet.

3. Vraťme se ale k automatizaci OLE. Jsou dva typy serverů automatizace OLE: servery in-proces a lokální servery. In-proces servery jsou DLL, které jsou zaváděny do adresního prostoru našeho programu. Lokální

servery jsou standardní aplikace obsahující server automatizace OLE. MS Word je příkladem lokálního serveru. Hlavním smyslem vytváření in-proces serverů je sdílení objektu zapsaného v jednom programovacím jazyku s aplikací zapsanou v jiném programovacím jazyku. Např. můžeme sdílet objekty Builderu s aplikací C++, Delphi nebo Visual Basicu prostřednictvím metod.

Klasickým příkladem automatizace OLE je řízení MS Wordu. Jestliže na našem počítači máme instalován Word 6.x nebo Word 7.x, potom k němu může přistupovat z aplikace Builderu zápisem následujícího kódu (jedná se o obsluhu stisku tlačítka; formulář obsahuje pouze tlačítko – nezapomeňte vložit hlavičkový soubor *OleAuto.hpp*):

```
Variant V;
V = CreateOleObject("Word.Basic");
V.OleProcedure("Insert", "Pozdrav z Builderu");
```

Tento kód vloží slova „Pozdrav z Builderu“ do existujícího dokumentu Wordu. Pro práci tohoto kódu musí být Word spuštěný s otevřeným dokumentem. Později se naučíme vyřešit situaci, kdy Word spuštěný není. Ve výše uvedeném kódu jsou tři klíčové prvky. Prvním je použití hlavičkového souboru *OleAuto*. Tento hlavičkový soubor obsahuje všechen kód pro zpracování automatizace zevnitř aplikací Builderu. Do C++ byl zaveden typ **Variant**, protože je používán při automatizaci OLE. Usnadňuje implementování OLE. V našem programu vytváříme tedy objekt automatizace OLE typu **Variant**:

```
V = CreateOleObject("Word.Basic");
```

Tento kód přiřazuje objekt OLE variantě V. Vytvářený objekt OLE je umístěn ve Wordu, je to server automatizace OLE. Třetí klíčový prvek v kódu je volání metody *Insert* Word Basicu:

```
V.OleProcedure("Insert", "Pozdrav z Builderu");
```

Insert není metoda nebo funkce C++, není ani součástí Windows API. Je součástí Wordu a díky automatizaci OLE ji můžeme volat přímo z aplikace Builderu.

Když voláme metodu *CreateOleObject*, předáváme ji řetězec. Tento řetězec obsahuje něco co nazýváme **ProgID** (identifikace programu). *Word.Basic* je **ProgID** pro server automatizace OLE v MS Word. **ProgID** je řetězec, který může být vyhledáván v registru a který ukazuje na CLSID. CLSID je unikátní číslo, které může být použito operačním systémem k odkazu na objekt OLE. Když voláme *CreateOleObject* pak Windows se podívá, zda Word je zaveden v paměti, a pokud není je zahájeno jeho zavádění. Jestliže je nalezeno rozhraní (ukazatel) na požadovaný objekt OLE, je vrácen ve tvaru výsledku volání *CreateOleObject*. Není vrácen ukazatel na Word samotný, ale ukazatel na objekt umístěný ve Wordu. Když máme rozhraní k objektu automatizace OLE ve Wordu, můžeme zahájit volání funkcí z objektu OLE. Můžeme používat funkce Word Basicu, jejich popis nalezneme v nápovědě Wordu. Jedná se asi o 200 funkcí a zahrnují příkazy pro otevírání dokumentů, ukládání dokumentů, formátování dokumentů, tisk dokumentů apod.

4. Jestliže chceme něco vložit do dokumentu, když Word není spuštěn, pak můžeme použít následující kód (začneme s vývojem nové aplikace a na formulář nazvaný **TOleWordForm** vložíme tlačítko, které nazveme *TalkToWord*; jako soukromou položku formuláře ještě vložíme **Variant V;**). K projektu přidáme další formulář nazvaný **TNotifyForm** obsahující pouze komponentu **Label**; uložíme jej do souboru *Notify*. Následuje výpis celé programové jednotky formuláře:

```
#include <vcl\vcl.h>
#pragma hdrstop
#include <OleAuto.hpp>
#include <memory>
#include <stdio.h>
#include <dir.h>
#include <io.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <except.h>
#include <cstring.h>
#include "main.h"
#include "notify.h"
#pragma resource "*.dfm"
TOleWordForm *OleWordForm;
char *current_directory(char *path);
__fastcall TOleWordForm::TOleWordForm(TComponent* Owner) : TForm(Owner)
{
}
#pragma warn -stv
void __fastcall TOleWordForm::TalkToWordClick(TObject *Sender)
{
```

```
  static HANDLE shFileMap = NULL;
  //Získání aktuálního adresáře a vytvoření řetězce jména souboru
  char curdir[MAXPATH];
  current_directory(curdir);
  AnsiString strFullFileName(curdir);
  strFullFileName = strFullFileName + "\\CBUILDER.DOC";
  //Vytvoření okna zpráv
  std::auto_ptr<TNotifyForm> pfrm(new TNotifyForm(this));
  pfrm->Label1->Caption = "Spouštění Microsoft Word...";
  pfrm->Update();
  try{
    //Testování, zda tato funkce automatizace již není spuštěna
    shFileMap=CreateFileMapping((HANDLE)0xffffffff, NULL, PAGE_READWRITE,
                                0, 4, "OLEWord2SharedData");
    if (shFileMap != NULL && GetLastError() == ERROR_ALREADY_EXISTS){
      string s("Tato funkce automatizace je již spuštěna.  "
               "Ukončení automatizace.");
      xmsg(s).raise();
    }
    V = CreateOleObject("Word.Basic");
    pfrm->Label1->Caption = "Otevírání nového souboru ..." ;
    pfrm->Update();
    V.OleProcedure("FileNew", "Normal");
    pfrm->Label1->Caption = "Vkládání textu ..." ;
    pfrm->Update();
    V.OleProcedure("Insert", "Drahá babičko, \n     Posílám ");
    V.OleProcedure("Insert", "Ti dopis.\n");
    pfrm->Label1->Caption = "Ukládání CBUILDER.DOC,password=cbuilder...";
    pfrm->Update();
    V.OleProcedure("FileSaveAs", strFullFileName, NULL, NULL,"cbuilder");
    pfrm->Label1->Caption = "Zavření souboru CBUILDER.DOC. . ." ;
    pfrm->Update();
    V.OleProcedure("FileClose", 1);
    pfrm->Label1->Caption = "Ukončení operace!";
    pfrm->Update();
  }
  catch(EOleSysError& e){
    char buf[256];
    sprintf(buf, "%s.\n\n%s", e.Message,
            "Tento program vyžaduje Microsoft Word.\n");
    ShowMessage(buf);
  }
  catch(EOleException& e){
    char buf[256];
    pfrm->Label1->Caption = "Byl získán odkaz na EOleException...";
    sprintf(buf, "%s.\n\n%s.",
            e.Message,
            "Heslo tohoto souboru je \"cbuilder\".\n"
            "Nový soubor nemůže být uložen.");
    V.OleProcedure("AppHide");
    ShowMessage(buf);
    V.OleProcedure("AppShow");
  }
  catch(xmsg& e){
    pfrm->Label1->Caption = "Word vidí již vytvořený CBUILDER.DOC...";
    pfrm->Update();
    V.OleProcedure("AppHide");
    ShowMessage(e.why().c_str());
    V.OleProcedure("AppShow");
  }
  catch(...){}
  CloseHandle(shFileMap);
}
#pragma warn .stv
#pragma warn -sig
```

```
char *current_directory(char *path)
{
  strcpy(path, "X:\\");
  path[0] = 'A' + getdisk();
  getcurdir(0, path+3);
  return(path);
}
#pragma warn .sig
```

Zajímavé na tomto kódu je, že nikdy nezpůsobí zobrazení Wordu na obrazovce. Word je pouze dočasně zaveden do paměti, ale stále je neviditelný. V okamžiku, kdy proměnná **V** se dostane mimo rozsah platnosti, nebo když ji nastavíme na *varNothing*, je Word opět zastaven. Jediným důkazem, že výše uvedený kód pracoval, je vytvoření souboru CBUILDER.DOC. Jestliže tento soubor otevřeme a prohlédneme si jeho obsah, zjistíme, že byl vytvořen našim programem. Klíčové příkazy jsou zvýrazněny. První klíčový příkaz je volání *CreateOleObject*. Tento příkaz spouští Word, pokud již není spuštěn. Pokud není spuštěn, je mu předán parametr `Automation`. Word ví, že při spuštění s tímto parametrem se má spustit na pozadí bez svého zobrazení. V tomto případě je také sám automaticky uzavřen. V uvedeném kódu si také můžeme povšimnout, že odřádkování můžeme zadat vložením znaku \n do textu.

Příkazy *FileNew* a *FileSaveAs* jsou standardní procedury Word Basicu, se kterými se můžeme seznámit v nápovědě Wordu. Jestliže se podíváme do nápovědy na *FileSaveAs* uvidíme následující výpis:

```
FileSaveAs [Name = text][, .Format = number] [, .LockAnnot = number]
[, .Password = text] [, .AddToMru = number] [, .WritePassword = text]
[, .RecommendReadOnly = number] [, .EmbedFonts = number]
[, .NativePictureFormat = number] [, .FormsData = number]
[, .SaveAsAOCELetter = number]
```

Jak vidíme, *FileSaveAs* přebírá celkem 11 parametrů nazvaných Name, Format atd. V našem programu jsme ale místo těchto 11 parametrů použili pouze čtyři. To je umožněno tím, že Word podporuje volitelné parametry.

Prostudujte si tuto aplikaci a zjistěte jak pracuje.

5. Builder umožňuje vytvářet pole prvků typu **Variant**, které jsou verzí „bezpečných polí“ použitých v automatizaci OLE. Pole **Variant** je implementace „bezpečných polí“ v Builderu. Tato pole se označují jako bezpečná pole, protože obsahují informaci o počtu rozměrů a o mezích každého rozměru. Soubor nazvaný OLEAUTO32.DLL obsahuje řadu volání *SafeArrayX* pro manipulaci s těmito poli. Builder zapouzdřuje volání *SafeArrayX* několika funkcemi. Jejich použití si ukážeme v následující aplikaci. Na formulář vložíme dvě tlačítka (s texty *Jednorozměrné pole* a *Dvourozměrné pole*). Obsluha stisku prvního tlačítka je tvořena příkazy:

```
AnsiString S;
int I;
S = "";
Variant MyVariant(OPENARRAY(int,(0,5)),varInteger);
for(I=0; I <= 5; I++){
  MyVariant.PutElement(I*2,I);
}
for(I=0; I <= 5; I++){
  S = S + " " + AnsiString(MyVariant.GetElement(I));
}
ShowMessage(S);
```

a obsluha druhého tlačítka příkazy:

```
AnsiString S;
int I,J;
S = "";
Variant MyVariant(OPENARRAY(int,(0,5,0,5)),varInteger);
for(I=0; I <= 5; I++){
  for(J=0; J <= 5; J++){
    MyVariant.PutElement(I*J,I,J);
  }
}
for(I=0; I <= 5; I++){
  for(J=0; J <= 5; J++){
    S = S + " " + AnsiString(MyVariant.GetElement(I,J));
  }
  S = S + "\n";
```

```
}
ShowMessage(S);
```

Aplikaci vyzkoušejte a prostudujte si, jak můžeme pracovat s těmito poli. První parametr konstruktoru definuje rozměry pole a druhý parametr definuje typ proměnných uložených v poli. Jako druhý parametr nelze použít *varString*, v tomto případě lze použít *varOleStr*. Pole **Variant** může pomocí metody *ArrayRedim* změnit své rozměry:

```
void __fastcall ArrayRedim(int highBound);
```

Pro zjištění počtu rozměrů pole a mezí každého rozměru, můžeme použít metody *ArrayHighBound*, *ArrayLowBound* a *ArrayDimCount*.

```
int __fastcall ArrayDimCount() const;
int __fastcall ArrayLowBound(const int dim = 1) const;
int __fastcall ArrayHighBound(const int dim = 1) const;
```

Jestliže chceme urychlit proces zpracování polí, můžeme použít metody *ArrayLock* a *ArrayUnlock*. První z nich vrací ukazatel na data uložená v poli. Obecně, *ArrayLock* přebírá pole **Variant** a vrací standardní pole. Pro toto použití, musí být pole deklarováno s jedním ze standardních typů, jako int, bool, String, nebo float. Typ použitý v poli **Variant** a typ použitý v poli C++ musí být identický pro všechny prvky pole. Následuje příklad použití *ArrayLock* a *ArrayUnlock*:

```
const int HighVal = 12;
Variant __fastcall TLockingVariantForm::GetArray(){
  Variant V;
  int I, J;
  Variant V(OPENARRAY(int,(0,HighVal,0,HighVal)),varInteger);
  for (I = 0; I < HighVal; I++){
    for (J = 0; J < HighVal; J++){
      V.PutElement(I+J,I,J);
    }
  }
  return V;
}
void __fastcall TLockingVariantForm::bLockAryClick(TObject *Sender){
  int I, J;
  Variant V;
  int* Data;
  V = GetArray();
  Data = (int*) V.ArrayLock();
  try{
    for (I = 0; I < HighVal; I++){
      for (J = 0; J < HighVal; J++){
        Grid->Cells[I][J] = IntToStr(Data[I+J]);
      }
    }
  }
  catch(...){
    V.ArrayUnlock();
  }
  V.ArrayUnlock();
}
```

Tento kód nejprve uzamyká pole, a potom zpřístupňuje přes ukazatel standardní pole. Nakonec po ukončení operací uvolňuje pole. Nesmíme zapomenout volat *ArrayUnlock* po ukončení práce s daty v poli.

Jednou z velmi užitečných činností pro použití polí **Variant** je přenos binárních dat na a ze serveru. Jestliže máme binární soubory (např. WAV nebo AVI), můžeme je přenášet mezi naším programem a OLE serverem pomocí polí **Variant**. Toto je ideální situace pro použití *ArrayLock* a *ArrayUnlock*. V tomto případě musíme jako druhý parametr konstruktoru pole použít *varByte*. Budeme tedy pracovat přímo s polem slabik. Musíme pamatovat na to, že pole **Variant** je užitečné pouze ve speciálních případech. Je to velmi užitečný nástroj, obzvláště při volání objektů OLE. Pracuje ale pomaleji něž standardní pole C++, a je tedy vhodné je používat pouze tehdy, když je to nezbytné.

6. V tomto bodě se naučíme vytvářet jednoduchý server automatizace OLE použitím vestavěných tříd Builderu. Budeme postupovat v těchto krocích: Začneme s vývojem nové aplikace (zvolíme **New | Application**). Dále zvolíme **File | New** a na stránce **New** vybereme *Automation Object*. V dialogu zobrazeném po volbě *Automation Object* dáme třídě objektu jméno (např. *TMyAuto*), nastavíme **Instancing** na *Multiple Instance*, zadáme krátký popis a stiskneme OK. Dále uložíme novou programovou jednotku pod svým vlastním jménem,

např. MYAUTO.CPP. Po provedení výše uvedených kroků, zvolíme **Run | Parametres**, zadáme `/regserver` a spustíme aplikaci. Je proveden bezprostřední návrat, bez zobrazení hlavního formuláře. Můžeme nyní odstranit `/regserver` z dialogového okna **Parameters**. Registrační proces popsaný výše je nutno provést pouze jednou. Po jeho provedení je objekt registrován v systémovém registru. Jestliže chceme objekt odstranit, musíme naši aplikaci spustit s parametrem `/unregserver`. Jestliže se nyní podíváme do kódu programové jednotky *MyAuto*, nalezneme zde následující deklaraci:

```
TAutoClassInfo AutoClassInfo;
AutoClassInfo.AutoClass = __classid(TMyAuto);
AutoClassInfo.ProgID = "Project1.MyAuto";
AutoClassInfo.ClassID = "{69BB2891-6A5E-11D1-94F1-E01EB823473E}";
AutoClassInfo.Description = "Pokus";
AutoClassInfo.Instancing = acMultiInstance;
Automation->RegisterClass(AutoClassInfo);
```

Klíčovými prvky zde jsou **ProgID** a **ClassID**. Když spustíme program s parametrem `/regserver`, tyto klíče jsou přidány do registrační databáze. **ProgID** je alfanumerická přezdívka pro **ClassID**. Přezdívka je užitečná pro snadnější odkazování se na objekt, než přímé používání **ClassID**. K ověření existence těchto klíčů, spustíme REGEDIT.EXE z adresáře Windows. Jestliže program spustíme a jestliže máme úspěšně přidány položky v registrační databázi, potom můžeme úspěšně kompletovat server automatizace OLE.

Ve struktuře *AutoClassInfo* zobrazené výše vidíme, že server *Project1.MyAuto* je *MultiInstance*. Mohou zde být tyto možnosti: *acMultiInstance*, *acSingle* nebo *acInternal*. Jestliže více klientů může bezpečně najednou přistupovat k našemu serveru, musíme jeho typ deklarovat jako *acMultiInstance*. Pokud k němu může přistupovat najednou pouze jeden klient, může být typu *acSingle*.

Funkce, které chceme zveřejnit z našeho serveru automatizace OLE, musíme přidat do sekce automatizace našeho potomka **TAutoObject** (následuje výpis celého hlavičkového souboru; je zde i další rozšíření):

```
#ifndef myautoH
#define myautoH
#include <OleAuto.hpp>
class TMyAuto : public TAutoObject{
  private:
         AnsiString FMyProp;
         AnsiString __fastcall GetMyProp();
         void __fastcall SetMyProp(AnsiString S);
  public:
         __fastcall TMyAuto();
  __automated:
        void __fastcall ShowDialog();
        __property AnsiString MyProp = {read=GetMyProp, write=SetMyProp};
};
#endif
```

Kód uvedený výše má jednu veřejnou metodu a jednu vlastnost (slouží k nastavení hodnoty řetězce a metoda zobrazuje řetězec v dialogovém okně) přístupnou z kontejnerů automatizace. Implementace této metody nevyžaduje žádný speciální kód (následuje výpis celé jednotky formuláře):

```
int Initialization();
static int Initializer = Initialization();
#undef RegisterClass
#include "myauto.h"
#include <Dialogs.hpp>
__fastcall TMyAuto::TMyAuto() : TAutoObject(){
}
void __fastcall TMyAuto::ShowDialog(){
        if (FMyProp == ""){
           FMyProp = "This object has a property called MyProp";
           ShowMessage(FMyProp);
        }
}
void __fastcall TMyAuto::SetMyProp(AnsiString S){
                ShowMessage("In SetMyProp()");
                ShowMessage(S);
        FMyProp = S;
}
AnsiString __fastcall TMyAuto::GetMyProp(){
                ShowMessage("In GetMyProp()");
```

```
       return FMyProp;
}
void RegisterMyAuto(){
      TAutoClassInfo AutoClassInfo;
      AutoClassInfo.AutoClass = __classid(TMyAuto);
      AutoClassInfo.ProgID = "AUTOPROJ.MyAuto";
      AutoClassInfo.ClassID = "{FE67CF61-2EDD-11CF-B536-0080C72EFD43}";
      AutoClassInfo.Description = "Sam";
      AutoClassInfo.Instancing = acMultiInstance;
      Automation->RegisterClass(AutoClassInfo);
}
int Initialization(){
  RegisterMyAuto();
  return 0;
}
```

Pro použití v deklaracích metod nebo vlastností v sekci automatizace jsou přípustné následující typy: Currency, double, int, float, SmallInt, String, TDateTime, Variant a WordBool. Ostatní typy nelze použít. Když měníme rozhraní existujícího serveru automatizace, musíme vždy dbát na zpětnou kompatibilitu. Nemůžeme tedy odstraňovat vlastnosti nebo metody (bez vzniku chyby v existujících kontejnerech). Můžeme tedy pouze přidávat vlastnosti nebo metody k již existujícím. Jestliže modifikujeme rozhraní bez zajištění zpětné kompatibility, je vhodné objekt přejmenovat. Následuje seznam pravidel, které musíme v sekci automatizace dodržet: Deklarace vlastností mohou obsahovat pouze přístupové specifikátory (**read** a **write**). Ostatní specifikátory jsou nepřípustné. Přístupové specifikátory musí používat metody. Identifikátory položek nejsou dovoleny. Metody přístupu k vlastnostem a metody musí používat registrové konvence volání. Přetypování vlastností není dovoleno. Metody mohou být virtuální, ale ne dynamické. Přetypování metod je povoleno. Deklarace vlastnosti nebo metody může obsahovat volitelnou direktivu *dispid*, která musí být následována konstantním výrazem typu int, který udává identifikátor vyřízení vlastnosti nebo metody. Jestliže *dispid* není přítomen, překladač automaticky vybere číslo o jednotku větší než největší použitý identifikátor vyřízení v třídě a jejich potomcích. Specifikací již použitého identifikátoru vyřízení způsobí chybu.

Následuje kód projektového souboru serveru automatizace:

```
#include <vcl\vcl.h>
#pragma hdrstop
USERES("AUTOPROJ.res");
USEFORM("MAIN.CPP", Form1);
USEUNIT("MYAUTO.CPP");
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
        Application->Initialize();
        Application->CreateForm(__classid(TForm1), &Form1);
        Application->Run();
        return 0;
}
```

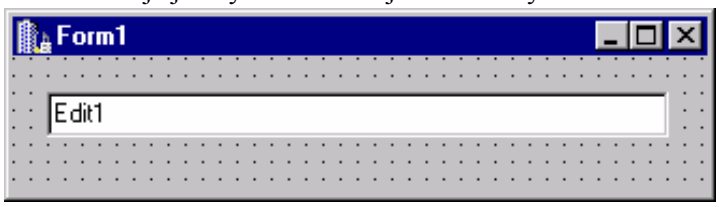
Tento kód aktualizuje registraci, jestliže předáme `/regserver` nebo `/unregserver` jako parametr aplikace. Nesmíme zapomenout, že `/regserver` musíme zadat alespoň jednou a vytvořit tak odpovídající prvek v registrační databázi. Nyní, když máme kompletní a funkční server automatizace, můžeme jej snadno otestovat vytvořením samostatné aplikace kontejneru automatizace OLE, který používá hlavičkový soubor *OleAuto.hpp*, obsahuje tlačítko a několik dalších řádků kódu:

```
#include <vcl.h>
#pragma hdrstop
#include "Unit1.h"
#include <OleAuto.hpp>
#pragma resource "*.dfm"
TForm1 *Form1;
__fastcall TForm1::TForm1(TComponent* Owner)
  : TForm(Owner)
{
}
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  Variant V;
  try{
```

```
    V = CreateOleObject("AutoProj.MyAuto");
    V.OleProcedure("ShowDialog");
    V.OlePropertySet("MyProp",
                     "This string was passed to the MyProp "
                     "property of AutoProj by TestAp");
    V.OleProcedure("ShowDialog");
  }
  catch(EOleSysError& e){
    char buf[256];
    sprintf(buf, "%s.\n\n%s", e.Message.c_str(),
     "Make sure you compile AutoProj first and run it either from\n"
      "the commmand line with the /regserver switch or from C++Builder\n"
      "after you set the Run|Parameters dialog to /regserver.");
    ShowMessage(buf);
  }
}
```

Když spustíme tuto aplikaci a stiskneme tlačítko, které volá výše uvedený kód, potom server automatizace je automaticky zaveden a je volána metoda *ShowDialog*.

7. Následující aplikace ukazuje jak vytvořit složitější server. Vytvoříme následující formulář:



a vytvoříme potřebné obsluhy událostí. Následuje výpis projektového souboru:

```
#include <vcl.h>
#pragma hdrstop
USEFORM("auto1.cpp", Form1);
USEUNIT("auto2.cpp");
USERES("autosrv.res");
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
    Application->Initialize();
    Application->CreateForm(__classid(TForm1), &Form1);
    Application->Run();
    return 0;
}
```

Vidíme, že tato aplikace se skládá ze jednoho formuláře (obsahuje pouze komponentu **Edit**) a jedné samostatné programové jednotky. Následuje výpis samostatné programové jednotky:

```
#include <vcl.h>
#pragma hdrstop
#undef RegisterClass
#include "Auto2.h"
#include "Auto1.h"
int Initialization();
static int Initializer = Initialization();
__fastcall ButtonServer::ButtonServer() : TAutoObject()
{
}
String __fastcall ButtonServer::GetEditStr()
{
  return Form1->Edit1->Text;
}
void __fastcall ButtonServer::SetEditStr(AnsiString NewVal)
{
  Form1->Edit1->Text = NewVal;
}
int __fastcall ButtonServer::GetEditNum()
{
  int val;
  sscanf(Form1->Edit1->Text.c_str(), "%d", &val);
  return val;
```

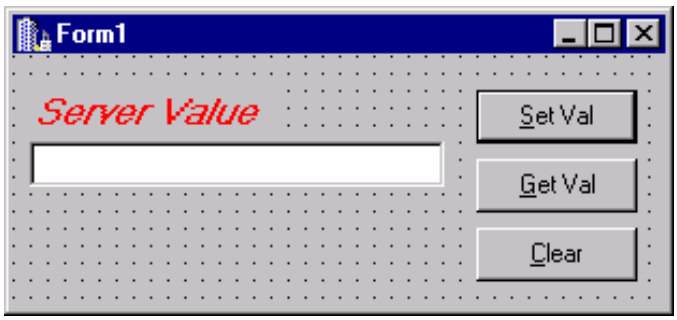
```
}
void __fastcall ButtonServer::SetEditNum(int NewVal)
{
  Form1->Edit1->Text = NewVal;
}
void __fastcall ButtonServer::Clear()
{
  Form1->Edit1->Text = "";
}
void __fastcall ButtonServer::SetThreeStr(AnsiString s1, AnsiString s2,
                                          AnsiString s3)
{
  Form1->Edit1->Text = s1 + ", " + s2 + ", " + s3;
}
void __fastcall ButtonServer::SetThreeNum(int n1, int n2, int n3)
{
  AnsiString s1(n1), s2(n2), s3(n3);
  Form1->Edit1->Text = s1 + ", " + s2 + ", " + s3;
}
void __fastcall RegisterButtonServer()
{
  TAutoClassInfo AutoClassInfo;
  AutoClassInfo.AutoClass = __classid(ButtonServer);
  AutoClassInfo.ProgID = "BCBAutoSrv.EditServer";
  AutoClassInfo.ClassID = "{61E124E1-C869-11CF-9EA7-00A02429B18A}";
  AutoClassInfo.Description="Borland C++Builder AutoSrv Example Server Class";
  AutoClassInfo.Instancing = acMultiInstance;
  Automation->RegisterClass(AutoClassInfo);
}
int Initialization()
{
  RegisterButtonServer();
  return 0;
}
```

Podle předchozího obrázku a výpisu vytvořte tuto aplikaci. Při prvním spuštění této aplikace musíme jako parametr předat `/regserver` ve volbě **Run | Parametres**. To umístí **ProgID** *BCBAutoSrv.EditServer* do registru. Spuštěním *RegEdit* si můžeme ověřit, zda registrace proběhla úspěšně.

K řízení aplikace serveru **AutoSrv** spustíme program **AutoCon**, který má následující formulář (a dále uvedený hlavičkový soubor a programovou jednotku):

```
#ifndef Auto1H
#define Auto1H
#include <OleAuto.hpp>
#include <StdCtrls.hpp>
#include <Dialogs.hpp>
#include <Forms.hpp>
#include <Controls.hpp>
#include <Graphics.hpp>
#include <Classes.hpp>
#include <SysUtils.hpp>
#include <Messages.hpp>
#include <Windows.hpp>
#include <System.hpp>
class TForm1 : public TForm
{
__published:
    TEdit *Edit1;
    TButton *Button1;
    TButton *Button2;
    TButton *Button3;
    TLabel *Label1;
    void __fastcall Button1Click(TObject *Sender);
    void __fastcall Button2Click(TObject *Sender);
    void __fastcall Button3Click(TObject *Sender);
private:
```

```
    Variant AutoServer;
public:
    virtual __fastcall TForm1(TComponent *Owner);
};
extern TForm1 *Form1;
#endif
```




```
#include <vcl.h>
#pragma hdrstop
#include "auto1.h"
#pragma resource "*.dfm"
TForm1 *Form1;
__fastcall TForm1::TForm1(TComponent *Owner) : TForm(Owner)
{
  try
  {
    AutoServer = CreateOleObject("BCBAutoSrv.EditServer");
  }
  catch (...)
  {
    ShowMessage("Please build and run the AutoSrv example before this one.");
    Application->Terminate();
  }
}
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  AutoServer.OlePropertySet("EditStr", Edit1->Text);
}
void __fastcall TForm1::Button2Click(TObject *Sender)
{
  Edit1->Text = AutoServer.OlePropertyGet("EditStr");
}
void __fastcall TForm1::Button3Click(TObject *Sender)
{
  AutoServer.OleProcedure("Clear");
}
```

V okamžiku spuštění programu **AutoCon**, zjistíme, že aplikace **AutoSrv** je také zavedena do paměti. Aplikace **AutoCon** slouží k řízení serveru **AutoSrv**. Prostudujte si tuto aplikaci a zjistěte jak pracuje. Vyzkoušejte spolupráci obou aplikací.

8. Programová jednotka **OleAuto** obsahuje objekt nazvaný **Automatization**. Tento objekt je používán pouze servery automatizace. Není důležitý pro kontejnery automatizace. Zde jsou pouze ty vlastnosti a událost objektu **TAutomatization**, které by mohly být užitečné:

```
__property bool IsInprocServer;
__property TStartMode StartMode;
__property TLastReleaseEvent OnLastRelease;
```

Vlastnost **IsInprocServer** určuje zda náš objekt automatizace je In-proces server nebo lokální server. Vlastnost **StartMode** vrací jednu z následujících hodnot: *smStandAlone*, *smAutomation*, *smRegServer*, *smUngegServer*. Význam těchto hodnot je uveden v následujíc tabulce:

Hodnota	Význam
smStandAlone	Uživatel spouští aplikaci.
SmAutomation	Windows spouští aplikaci pro vytvoření objektu OLE.
SmRegServer	Aplikace je spuštěna k registraci jednoho nebo více OLE objektů.
SmUnregServer	Aplikace je spuštěna k odregistraci jednoho nebo více OLE objektů.

Událost **OnLastRelease** je generována, když server zavedený systémem pro automatizaci již není potřeba. To se vyskytne, když všechny klienti uvolnily všechny objekty OLE, vytvořené serverem. Builder obvykle automaticky uvolní server, když již není zapotřebí, ale toto můžeme předefinovat. **OnLastRelease** má parametr typu Bool volaný odkazem nazvaný *ShutDown*, který je implicitně *True*. Nastavením *ShutDown* na *False* předefinuje implicitní chování Builderu a zachová server v paměti, přestože již není zapotřebí.

Server In-proces je dynamicky sestavitelná knihovna, která exportuje objekty automatizace. Objekty automatizace získané z DLL, jsou součástí stejného procesu Windows jako klientská aplikace. Servery In-proces jsou užitečné při vytváření programových modulů sdílených několika aplikacemi, které mohou být zapsány v různých programových jazycích. Mají výhodu, že se spouštějí ve stejném adresovém prostoru jako volající aplikace (umožňuje to aby volání probíhalo přes hranice aplikace). Počáteční kroky při vytváření In-proces serveru automatizace OLE se neliší od vytváření jiných DLL. Začneme vytvořením DLL. Můžeme přidat jednotky a formuláře jako obvykle a můžeme přidat objekty OLE. Přidáme také hlavičkový soubor *OleAuto*.

12. Vícevláknové aplikace

1. Na závěr si ještě ukážeme použití vícevláknové aplikace. Předpokládejme, že chceme demonstrovat postup a rychlost řazení pole hodnot několika různými metodami. Porovnání rychlosti řazení různými metodami provedeme nejlépe tak, že tyto jednotlivé metody naprogramujeme a spustíme je současně (toto nám umožní vícevláknová aplikace). Musíme vytvořit více vláknový objekt, který je potomkem třídy **TTherad**. V našem případě vytvoříme třídu **TSortThread**. Začneme vývojem nové aplikace (zvolíme **File | New Application**). Pro vytvoření vícevláknového objektu zvolíme **File | New** a na stránce **New** vybereme *Thread object*. Tím vytvoříme novou programovou jednotku pro uložení vícevláknového objektu (musíme zadat jméno vytvářené třídy; použijeme **TSortThread**). Do této jednotky doplníme deklaraci naši třídy podle následujícího výpisu (jednotku také přejmenujeme na *SortThd.cpp*). Začneme výpisem hlavičkového souboru:

```
#ifndef SortThdH
#define SortThdH
#include <ExtCtrls.hpp>
#include <Graphics.hpp>
#include <Classes.hpp>
#include <System.hpp>
extern void __fastcall PaintLine(TCanvas *Canvas, int i, int len);
class TSortThread : public TThread
{
private:
    TPaintBox *FBox;
    int *FSortArray;
    int FSize;
    int FA;
    int FB;
    int FI;
    int FJ;
    void __fastcall DoVisualSwap(void);
protected:
    virtual void __fastcall Execute(void);
    void __fastcall VisualSwap(int A, int B, int I, int J);
    virtual void __fastcall Sort(int *A, const int A_Size) = 0;
public:
    __fastcall TSortThread(TPaintBox *Box, int *SortArray,
        const int SortArray_Size);
};
class TBubbleSort : public TSortThread
{
protected:
    virtual void __fastcall Sort(int *A, const int A_Size);
public:
    __fastcall TBubbleSort(TPaintBox *Box, int *SortArray,
        const int SortArray_Size);
};
class TSelectionSort : public TSortThread
{
protected:
    virtual void __fastcall Sort(int *A, const int A_Size);
```

```
public:
    __fastcall TSelectionSort(TPaintBox *Box, int *SortArray,
                              const int SortArray_Size);
};
class TQuickSort : public TSortThread
{
protected:
    void __fastcall QuickSort(int *A, const int A_Size, int iLo,
                              int iHi);
    virtual void __fastcall Sort(int *A, const int A_Size);
public:
    __fastcall TQuickSort(TPaintBox *Box, int *SortArray,
        const int SortArray_Size);
};
#endif

#include <vcl.h>
#pragma hdrstop
#include "sortthd.h"
void __fastcall PaintLine(TCanvas *Canvas, int I, int Len)
{
  TPoint points[2];
  points[0] = Point(0, I*2+1);
  points[1] = Point(Len, I*2+1);
  Canvas->Polyline(EXISTINGARRAY(points));
}
__fastcall TSortThread::TSortThread(TPaintBox *Box, int *SortArray,
    const int SortArray_Size) : TThread(False)
{
  FBox = Box;
  FSortArray = SortArray;
  FSize = SortArray_Size + 1;
  FreeOnTerminate = True;
}
/*Jelikož DoVisualSwap používá komponenty VCL nemůže být nikdy volána
  přímo vláknem. DoVisualSwap musí být volána předáním metodě Synchronize
  způsobující, že DoVisualSwap bude provedeno hlavním vláknem VCL. Viz
  volání Synchronize v metodě VisualSwap. */
void __fastcall TSortThread::DoVisualSwap()
{
  TCanvas *canvas;
  canvas = FBox->Canvas;
  canvas->Pen->Color = TColor(clBtnFace);
  PaintLine(canvas, FI, FA);
  PaintLine(canvas, FJ, FB);
  canvas->Pen->Color = clRed;
  PaintLine(canvas, FI, FB);
  PaintLine(canvas, FJ, FA);
}
/*VisusalSwap usnadňuje používání DoVisualSwap. Parametry jsou
  překopírovány do proměnných instance, čímž je zpřístupňuje hlavnímu
  vláknu VCL když provádí DoVisualSwap */
void __fastcall TSortThread::VisualSwap(int A, int B, int I, int J)
{
  FA = A;
  FB = B;
  FI = I;
  FJ = J;
  Synchronize(DoVisualSwap);
}
/* Metoda Execute je volána při spouštění vlákna */
void __fastcall TSortThread::Execute()
{
  Sort(FSortArray, FSize-1);
}
```

```
__fastcall TBubbleSort::TBubbleSort(TPaintBox *Box, int *SortArray,
  const int SortArray_Size) : TSortThread(Box, SortArray, SortArray_Size)
{
}
void __fastcall TBubbleSort::Sort(int *A, int const AHigh)
{
  int I, J, T;
  for (I=AHigh; I >= 0; I--)
    for (J=0; J<=AHigh-1; J++)
      if (A[J] > A[J + 1])
      {
        VisualSwap(A[J], A[J + 1], J, J + 1);
        T = A[J];
        A[J] = A[J + 1];
        A[J + 1] = T;
        if (Terminated)
          return;
      }
}
__fastcall TSelectionSort::TSelectionSort(TPaintBox *Box, int *SortArray,
  const int SortArray_Size) : TSortThread(Box, SortArray, SortArray_Size)
{
}
void __fastcall TSelectionSort::Sort(int *A, int const AHigh)
{
  int I, J, T;
  for (I=0; I <= AHigh-1; I++)
    for (J=AHigh; J >= I+1; J--)
      if (A[I] > A[J])
      {
        VisualSwap(A[I], A[J], I, J);
        T = A[I];
        A[I] = A[J];
        A[J] = T;
        if (Terminated)
          return;
      }
}
__fastcall TQuickSort::TQuickSort(TPaintBox *Box, int *SortArray,
  const int SortArray_Size) : TSortThread(Box, SortArray, SortArray_Size)
{
}
void __fastcall TQuickSort::QuickSort(int *A, int const AHigh, int iLo,
                                      int iHi)
{
  int Lo, Hi, Mid, T;
  Lo = iLo;
  Hi = iHi;
  Mid = A[(Lo+Hi)/2];
  do
  {
    if (Terminated)
      return;
    while (A[Lo] < Mid)
        Lo++;
    while (A[Hi] > Mid)
        Hi--;
    if (Lo <= Hi)
    {
      VisualSwap(A[Lo], A[Hi], Lo, Hi);
      T = A[Lo];
      A[Lo] = A[Hi];
      A[Hi] = T;
      Lo++;
      Hi--;
```

```
    }
  }
  while (Lo <= Hi);
  if (Hi > iLo)
    QuickSort(A, AHigh, iLo, Hi);
  if (Lo < iHi)
    QuickSort(A, AHigh, Lo, iHi);
}
void __fastcall TQuickSort::Sort(int *A, int const AHigh)
{
  QuickSort(A, AHigh, 0, AHigh);
}
```

V této jednotce jsou jednotlivé řadící metody deklarovány jako třídy odvozené od **TSortThread**. Formulář aplikace zvětšíme a nastavíme u něj tyto vlastnosti: **BorderStyle** na *bsDialog* a **Caption** na *Demonstrace řazení*. Na formulář přidáme tři komponenty **Bevel** a nastavíme u nich vlastnosti **Top** na 24, **Width** na 177 a **Height** na 233. U první z nich nastavíme **Left** na 8, u druhé na 192 a u třetí na 376. Do každé z těchto komponent vložíme komponentu **PaintBox** a nastavíme u nich stejné hodnoty vlastností jako u komponent **Bevel**. Levou z těchto komponent nazveme *BubbleSortBox*, prostřední *SelectionSortBox* a pravou *QuickSortBox*. Nad každou z těchto komponent vložíme **Label** s texty: *Bublinková metoda*, *Řazení výběrem* a *Quick Sort*. Do spodní části formuláře vložíme ještě tlačítko s textem *Začni řadit*. Tím je tento formulář hotov. Do deklarace formuláře vložíme další deklarace (viz následující výpis hlavičkového souboru formuláře):

```
#ifndef ThSortH
#define ThSortH
#include <StdCtrls.hpp>
#include <ExtCtrls.hpp>
#include <Dialogs.hpp>
#include <Forms.hpp>
#include <Controls.hpp>
#include <Graphics.hpp>
#include <Classes.hpp>
#include <SysUtils.hpp>
#include <Messages.hpp>
#include <Windows.hpp>
#include <System.hpp>
class TThreadSortForm : public TForm
{
__published:
    TButton *StartBtn;
    TPaintBox *BubbleSortBox;
    TPaintBox *SelectionSortBox;
    TPaintBox *QuickSortBox;
    TLabel *Label1;
    TBevel *Bevel1;
    TBevel *Bevel2;
    TBevel *Bevel3;
    TLabel *Label2;
    TLabel *Label3;
    void __fastcall BubbleSortBoxPaint(TObject *Sender);
    void __fastcall SelectionSortBoxPaint(TObject *Sender);
    void __fastcall QuickSortBoxPaint(TObject *Sender);
    void __fastcall FormCreate(TObject *Sender);
    void __fastcall StartBtnClick(TObject *Sender);
private:
    int ThreadsRunning;
    void __fastcall RandomizeArrays(void);
    void __fastcall ThreadDone(TObject *Sender);
public:
    void __fastcall PaintArray(TPaintBox *Box, const int *A,
                               const int A_Size);
    virtual __fastcall TThreadSortForm(TComponent *Owner);
};
typedef int TSortArray[115];
```

```
typedef TSortArray *PSortArray;
extern TThreadSortForm *ThreadSortForm;
extern bool ArraysRandom;
extern int BubbleSortArray[115];
extern int SelectionSortArray[115];
extern int QuickSortArray[115];
#endif
```

Následuje výpis jednotky formuláře:

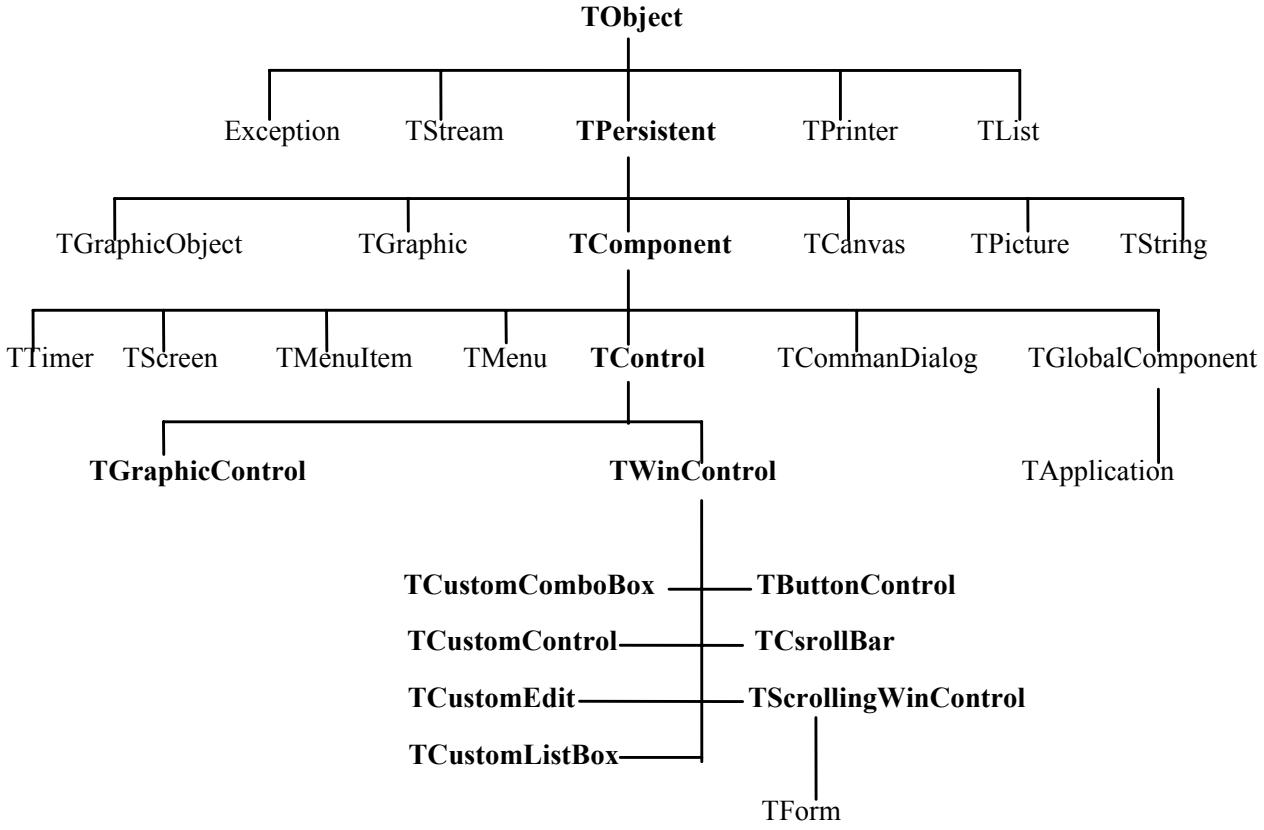
```
#include <vcl.h>
#pragma hdrstop
#include <stdlib.h>
#include "thsort.h"
#include "sortthd.h"
#pragma resource "*.dfm"
TThreadSortForm *ThreadSortForm;
Boolean ArraysRandom;
TSortArray BubbleSortArray, SelectionSortArray, QuickSortArray;
__fastcall TThreadSortForm::TThreadSortForm(TComponent *Owner)
  : TForm(Owner)
{
}
void __fastcall TThreadSortForm::PaintArray(TPaintBox *Box, int const *A,
  int const ASize)
{
  int i;
  TCanvas *canvas;
  canvas = Box->Canvas;
  canvas->Pen->Color = clRed;
  for (i=0; i < ASize; i++)
    PaintLine(canvas, i, A[i]);
}
void __fastcall TThreadSortForm::BubbleSortBoxPaint(TObject * /*Sender*/)
{
  PaintArray(BubbleSortBox, EXISTINGARRAY(BubbleSortArray));
}
void __fastcall TThreadSortForm::SelectionSortBoxPaint(TObject *
                                                       /*Sender*/)
{
  PaintArray(SelectionSortBox, EXISTINGARRAY(SelectionSortArray));
}
void __fastcall TThreadSortForm::QuickSortBoxPaint(TObject * /*Sender*/)
{
  PaintArray(QuickSortBox, EXISTINGARRAY(QuickSortArray));
}
void __fastcall TThreadSortForm::FormCreate(TObject * /*Sender*/)
{
  RandomizeArrays();
}
void __fastcall TThreadSortForm::StartBtnClick(TObject * /*Sender*/)
{
  TBubbleSort *bubble;
  TSelectionSort *selsort;
  TQuickSort *qsort;
  RandomizeArrays();
  ThreadsRunning = 3;
  bubble =new TBubbleSort(BubbleSortBox, EXISTINGARRAY(BubbleSortArray));
  bubble->OnTerminate = ThreadDone;
  selsort = new TSelectionSort(SelectionSortBox,
    EXISTINGARRAY(SelectionSortArray));
  selsort->OnTerminate = ThreadDone;
  qsort = new TQuickSort(QuickSortBox, EXISTINGARRAY(QuickSortArray));
  qsort->OnTerminate = ThreadDone;
  StartBtn->Enabled = False;
}
```

```
void __fastcall TThreadSortForm::RandomizeArrays()
{
  int i;
  if (! ArraysRandom)
  {
    Randomize();
    for (i=0; i < ARRAYSIZE(BubbleSortArray); i++)
      BubbleSortArray[i] = random(170);
    memcpy(SelectionSortArray, BubbleSortArray,
           sizeof(SelectionSortArray));
    memcpy(QuickSortArray, BubbleSortArray, sizeof(QuickSortArray));
    ArraysRandom = True;
    Repaint();
  }
}
void __fastcall TThreadSortForm::ThreadDone(TObject * /*Sender*/)
{
  ThreadsRunning--;
  if (! ThreadsRunning)
  {
    StartBtn->Enabled = True;
    ArraysRandom = False;
  }
}
```

Pokuste se vytvořit aplikaci podle tohoto výpisu a snažte se pochopit jednotlivé činnosti. Vyzkoušejte. Pokuste se zjistit jak pracují vícevláknové aplikace.

13. Úvod do vytváření komponent

1. Jednou z klíčových možností Builderu je možnost rozšiřování knihovny komponent. Všechny komponenty Builderu patří do objektové hierarchie nazvané knihovna vizuálních komponent (VCL). Na následujícím obrázku je uvedena zjednodušená hierarchie objektů tvořících VCL.



Typ **TComponent** je společný předek všech komponent VCL. **TComponent** poskytuje minimální vlastnosti a události nezbytné pro práci komponenty v Builderu. Různé větve knihovny poskytují další více specializované komponenty. Vytvářenou komponentu přidáme do VCL odvozením nového objektu od některého existujícího typu objektu v hierarchii. Komponenty jsou objekty a tvůrce komponent pracuje s objekty na jiné úrovni než uživatel komponent. Vytvoření nové komponenty vyžaduje odvození nového typu objektu. Jsou dva hlavní rozdíly mezi vytvářením komponent a používáním komponent. Při vytváření komponent musíme mít přístup k částem objektu, které jsou nepřístupné pro koncového uživatele a můžeme přidávat nové části (např. vlastnosti) ke svým komponentám. S ohledem na tyto rozdíly, je nutno dodržovat mnoho konvencí a brát ohled na používání našich komponent koncovými uživateli.

Komponenty jsou prvky programu s nimiž manipulujeme během návrhu. Vytváření nové komponenty znamená odvození typu objektu nové komponenty od existujícího typu. V následující tabulce jsou uvedeny různé typy komponent a typy objektů od kterých je odvozujeme.

K provedení	**Začneme od typu**
modifikace existující komponenty	libovolné existující komponenty, jako je **TButton** nebo **TListBox**, nebo od typu abstraktní komponenty jako je **TCustomListBox**
Vytváření původního ovladače	**TCustomControl**
Vytváření grafického ovladače	**TGraphicControl**
Použití existujícího ovladače Windows	**TWinControl**
Vytváření nevizuální komponenty	**TComponent**

Můžeme také odvodit jiné objekty, které nejsou komponentami, ale nemůžeme s nimi manipulovat na formuláři. Builder obsahuje několik těchto typů objektů, jako je **TINIFile** nebo **TFont**.

2. Nejjednodušší způsob vytvoření komponenty je začít od existující funkční komponenty a přizpůsobit ji. Novou komponentu můžeme odvodit od libovolné komponenty poskytnuté Builderem. Např. můžeme chtít změnit hodnotu implicitní vlastnosti jednoho ze standardních ovladačů.

   Některé ovladače, jako jsou komponenty seznamu a mřížky, mají mnoho variant na základní téma. V těchto případech Builder poskytuje abstraktní typ ovladače (se slovem „**Custom**“ ve svém jméně, jako je např.

TCustomGrid), který slouží k odvozování jednotlivých verzí. Např. můžeme potřebovat vytvořit speciální typ seznamu, který nemá některé vlastnosti standardního typu **TListBox**. Jelikož nelze odstranit vlastnost z typu předka, musíme svou komponentu odvodit od něčeho výše v hierarchii než je **TListBox**. Nemusíme začít od typu prázdného abstraktního ovladače a vytvářet všechny funkce seznamu, neboť VCL poskytuje **TCustomListBox**, který implementuje všechny vlastnosti potřebné pro seznam, ale všechny je nezveřejňuje. Když odvozujeme komponentu od některého z abstraktních typů, jako je **TCustomListBox**, zveřejníme ty vlastnosti, které chceme zpřístupnit ve své komponentě a ostatní ponecháme chráněné.

3. Při vytváření původního (nového) ovladače je důležité toto. Standardní ovladač je prvek, který je viditelný při běhu aplikace a obvykle s ním uživatel může pracovat. Tyto standardní ovladače jsou všechny potomky objektu **TCustomControl**. Když vytváříme původní ovladač (takový, který není svázán s žádným existujícím ovladačem), musíme jako počáteční bod použít **TCustomControl**. Klíčovým aspektem standardního ovladače je to, že má madlo okna, uložené ve vlastnosti nazvané *Handle*. Madlo okna umožňuje Windows „vědět o“ ovladači a mimo jiné umožňuje ovladači získat vstupní zaostření a předávat madlo funkcím Windows API (Windows potřebuje madlo k určení okna, se kterým má operovat). Jestliže náš ovladač nepotřebuje získat vstupní zaostření, můžeme jej vytvořit jako grafický ovladač, který nevyužívá systém zdrojů Windows. Všechny ovladače, které reprezentují standardní ovladače Windows, jako jsou tlačítka, seznamy a editační okna jsou potomky **TWinControl** (mimo **TLabel**, neboť tento ovladač nemůže získat vstupní zaostření).
4. Grafické ovladače jsou velmi podobné uživatelským ovladačům, ale nejsou odvozeny od ovladačů Windows, tj. Windows nic o grafických ovladačích neví. Nemají madlo okna a tedy nečerpají zdroje systému. Hlavním omezením grafických ovladačů je, že nemohou získat vstupní zaostření. Builder podporuje vytváření grafických uživatelských ovladačů prostřednictvím typu **TGraphicControl**. **TGraphicControl** je abstraktní typ odvozený od **TControl**. Přestože můžeme odvozovat ovladače od **TControl**, je vhodnější je odvozovat od třídy **TGraphicControl**, která poskytuje plátno na kreslení a zpracovává zprávu WM_PAINT a není tedy nutné přepisovat metodu *Paint*.
5. Windows má koncepci nazvanou třída okna, která se podobá koncepci objektově orientovaného programování objektů nebo tříd. Třída okna je množina informací sdílená mezi různými instancemi stejného typu oken nebo ovladačů ve Windows. Když vytváříme nový typ ovladače (obvykle nazývaný uživatelský ovladač) v tradičním programování Windows, definujeme novou třídu okna a registrujeme ji ve Windows. Můžeme také založit novou třídu okna na existující třídě. Jestliže chceme vytvořit uživatelský ovladač v tradičním programování Windows, musíme jej zapsat v DLL stejně jako standardní ovladače Windows a poskytnout k němu rozhraní. Pomocí Delphi můžeme vytvořit komponentu „obálkou“ okolo existující třídy Windows. Jestliže tedy máme knihovnu uživatelských ovladačů, které chceme používat ve svých aplikacích Delphi, můžeme vytvořit komponenty Delphi, ve kterých použijeme existující ovladače a odvodíme od nich nové ovladače a to stejně jako od jiných komponent. I když v této publikaci není uveden žádný příklad použití existujícího ovladače Windows, můžeme se s touto technikou seznámit v komponentách programové jednotky *StdCtls*, které reprezentují standardní ovladače Windows, např. **TEdit**.
6. Vytváření nevizuálních komponent. Abstraktní objekt **TComponent** je základním typem pro všechny komponenty. Nevizuální komponenty jsou pouze komponenty, které vytváříme přímo od **TComponent**. **TComponent** definuje všechny nezbytné vlastnosti a metody komponenty pro spolupráci s Návrhářem formuláře. Tedy libovolné komponenty odvozené od **TComponent** mají zabudované možnosti návrhu. Nevizuální komponenty jsou používány málo. Jejich využití je jako rozhraní pro nevizuální prvky programu (např. pro databázové prvky) a držení místa pro dialogová okna (např. souborová dialogová okna).
7. Je několik omezení na to, co můžeme vložit do komponenty. Nicméně, jsou jisté konvence, které je vhodné dodržovat, jestliže chceme udělat komponentu spolehlivou a snadno použitelnou pro její uživatele. Snad nejdůležitější princip tvorby komponent Builderu je nezbytnost odstranit závislosti. Jedna z věcí, která zjednodušuje použití komponent pro koncové uživatele v aplikaci, je skutečnost, že zde nejsou obecně žádná omezení na to, co s nimi můžeme udělat na jakémkoli daném místě svého kódu. Povaha komponent, předpokládá, že různí uživatelé je použijí ve svých aplikacích v rozličných kombinacích, pořadích a prostředcích. Měli bychom navrhovat své komponenty, aby jejich funkčnost v jakémkoli kontextu nebyla ničím podmíněna. Vytvořit komponenty, které nejsou závislé, možná zabere více času, je to ale vhodně využitý čas.
8. Kromě viditelného obrazu komponenty, se kterým uživatel manipuluje na formuláři při návrhu, jsou nejdůležitější atributy komponenty její **vlastnosti**, **události** a **metody**. **Vlastnosti** dávají uživatelům komponent iluzi nastavování nebo čtení hodnot proměnných v komponentě, zatímco tvůrce komponenty skrývá použité datové struktury a implementaci přístupu k hodnotám. Používání vlastností dává tyto výhody: Vlastnosti jsou přístupné během návrhu (to umožňuje uživatelům komponent nastavovat a měnit počáteční hodnoty vlastností bez zásahu do kódu), vlastnosti mohou testovat hodnoty nebo formáty, které jim uživatel přiřadí

(kontrolou uživatelova vstupu zabraňujeme chybám způsobeným nedovolenými hodnotami) a komponenta může na žádost vytvářet příslušnou hodnotu (častý typ programátorských chyb je odkaz na proměnnou, která nemá přiřazenou počáteční hodnotu; vytvořením hodnoty vlastnosti, můžeme zajistit, že hodnota vlastnosti je vždy přípustná). **Události** jsou propojením mezi výskyty určenými tvůrcem komponenty (např. akce myši nebo klávesnice) a kódem zapsaným uživatelem komponenty (obsluha události). Událost je možnost poskytnutá tvůrcem komponenty uživateli komponenty pro specifikaci kódu, který má být proveden v určitých případech. Uživatel komponenty může specifikovat obsluhu pro předdefinovanou událost a nemusí odvozovat vlastní komponentu. **Metody** jsou funkce zabudované v komponentě. Uživatel komponenty používá metody k přikázání komponentě, aby provedla specifickou akci nebo vrátila jistou hodnotu, která není vlastností. Metody jsou také užitečné pro aktualizaci několika svázaných vlastností jediným voláním. Protože metody vyžadují provedení kódu jsou dostupné pouze za běhu programu.

9. Builder odstraňuje namáhavou práci s grafikou Windows zaobalením různých grafických nástrojů do plátna. Plátno reprezentuje kreslící plochu okna nebo ovladače a obsahuje další objekty jako je pero, štětec a písmo. Plátno je něco jako kontext zařízení Windows, ale přebírá starost o řadu věcí za nás. Jestliže vytváříme grafickou aplikaci Windows, musíme se seznámit s typy požadavků grafického rozhraní Windows, jako jsou limity příslušných kontextů zařízení a obnovováním grafických objektů na jejich počáteční stav před jejich uvolněním. Když pracujeme s grafikou v Builderu, nemusíme tyto věci znát. Pro kreslení na formulář nebo komponentu, přistupujeme k vlastnosti *Canvas*. Jestliže chceme přizpůsobit pero nebo štětec, nastavíme barvu nebo styl. Když skončíme, Builder přebírá starost za uvolnění zdrojů. Máme stále plný přístup k GDI Windows, ale náš kód bude jednodušší a bude pracovat rychleji, jestliže použijeme plátno zabudované do komponent Builderu.
10. Dříve než můžeme komponentu použít při návrhu, musíme ji v Builderu registrovat. Registrace říká Builderu, na které stránce Palety komponent chceme komponentu zobrazit.
11. Při vytváření nové komponenty musíme provést několik kroků. Komponentu lze vytvořit dvěma způsoby: ručním vytvářením komponenty nebo pomocí Experta komponent. Ať již použijeme kterýkoli způsob, musíme komponentu, která má alespoň minimální funkčnost instalovat na Paletu komponent. Potom lze komponentě přidávat další funkce, aktualizovat paletu a pokračovat v testování. Nejsnadnějším způsobem vytváření nové komponenty je použití Experta komponent. Nicméně můžeme také provést stejné kroky ručně. Ruční vytváření komponenty probíhá v těchto krocích: vytvoření nové programové jednotky, odvození třídy komponenty, deklarování nového konstruktoru a registrace komponenty. Programová jednotka (včetně hlavičkového souboru) je samostatně překládaný modul kódu C++. Builder používá jednotky pro několik účelů. Každý formulář má svoji vlastní programovou jednotku a většina komponent (nebo logických skupin komponent) má také svou vlastní jednotku. Když vytváříme komponentu, můžeme pro komponentu vytvořit novou jednotku nebo přidat novou komponentu do existující jednotky. K vytvoření jednotky pro komponentu, zvolíme **File | New Unit**. Builder vytvoří nový soubor a otevře jej v Editoru kódu. Vytvořenou jednotku uložíme pod smysluplným jménem. Pro přidání komponenty k existující jednotce, zvolíme **File | Open** a vybereme zdrojový kód existující jednotky. Když přidáváme komponentu k existující jednotce, musíme si být jisti, že jednotka obsahuje pouze kód komponent. Přidání kódu komponent k jednotce, která obsahuje např. formulář, způsobí chybu. Jestliže máme novou nebo existující jednotku pro naši komponentu, můžeme odvodit objekt komponenty. Každá komponenta je třída odvozená od typu **TComponent**, od jednoho z více specializovaných potomků, jako je **TControl** nebo **TGraphicControl**, nebo od existujícího typu komponenty.

    K odvození třídy komponenty, přidáme deklaraci třídy do hlavičkového souboru jednotky, která má komponentu obsahovat. Pro vytvoření např. jednoduchého typu nevizuální komponenty ji odvodíme přímo od **TComponent** a do hlavičkového souboru naši jednotky přidáme následující definici typu:

```
class TNovaKomponenta : public TComponent
{
};
```

    Musíme také přidat příkazy vložení hlavičkových souborů, které jsou nutné pro novou komponentu. Většinou to budou direktivy:

```
#include <vcl\SysUtils.hpp>
#include <vcl\Controls.hpp>
#include <vcl\Classes.hpp>
#include <vcl\Forms.hpp>
```

    Nová komponenta se neliší od **TComponent**. Tvoří rámec, ve kterém budeme budovat svoji novou komponentu. Každá nová komponenta musí mít konstruktor, který přepisuje konstruktor třídy, od které komponentu odvozujeme. Když zapisujeme konstruktor pro novou komponentu, pak tento konstruktor musí vždy volat zděděný konstruktor. V deklaraci třídy deklarujeme virtuální konstruktor a to ve veřejné části třídy. Např.

```
class TNovaKomponenta : public TComponent
{
  public:
    __fastcall TNovaKomponenta(TComponent* Owner);
};
```

Do CPP souboru vložíme implementaci konstruktoru:

```
__fastcall TNovaKomponenta::TNovaKomponenta(TComponent* Owner)
    : TComponent(Owner)
{
}
```

Do konstruktoru přidáme kód, který chceme provést při vytváření komponenty

Nyní již můžeme registrovat **TNovaKomponenta**. Registrace komponenty je jednoduchý proces, který říká Builderu, které komponenty přidat do své knihovny komponent a na které stránce Palety komponent bude komponenta zobrazena. K registraci komponenty přidáme funkci nazvanou *Register* do souboru CPP a umístíme ji do jmenného prostoru. Jméno jmenného prostoru je jméno souboru komponenty bez přípony, ve kterém jsou s výjimkou prvního písmena všechna písmena malá. Např.

```
namespace Unit1
{
  void __fastcall Register()
  {
  }
}
```

V této funkci deklarujeme otevřené pole typu **TComponentClass**, které obsahuje registrované komponenty:

```
TComponentClass classes[1] = {__classid(TNovaKomponenta)};
```

V tomto příkladě pole obsahuje právě jednu komponentu, ale můžeme přidat další komponenty, které chceme registrovat. Dále voláme funkci **RegisterComponents**, pro každou registrovanou komponentu. Tato funkce přebírá tři parametry: jméno stránky palety konponent, pole tříd komponent a velikost pole tříd komponent zmenšenou o 1. Jestliže přidáváme komponentu k existující registraci, můžeme přidat novou komponentu k existujícímu příkazu nebo přidat nový příkaz, který volá **RegisterComponents**. Jedním voláním **RegisterComponents** můžeme registrovat i více komponent, jestliže jsou všechny na stejné stránce palety komponent. K registraci komponenty naznané **TNovaKomponenta** na stránce *Samples* Palety komponent tedy použijeme:

```
namespace Unit1
{
  void __fastcall Register()
  {
    TComponentClass classes[1] = {__classid(TNovaKomponenta)};
    RegisterComponents("Samples", classes, 0);
  }
}
```

Po registraci komponenty ji můžeme instalovat na Paletu komponent.

12. K vytvoření nové komponenty můžeme použít Experta komponent. Použití Experta komponent zjednodušuje počátek vytváření nové komponenty. Je zapotřebí zadat pouze jméno nové komponenty, typ předka a stránku Palety komponent, na které ji chceme zobrazit. Expert komponent provede stejné úlohy, jako kdybychom komponentu vytvářeli ručně. Expert komponent ale nedokáže přidat komponentu do již existující jednotky. K otevření Experta komponent zvolíme **Component | New…**. Po vyplnění požadovaných údajů stiskneme OK. Builder vytvoří novou jednotku obsahující deklaraci typu, funkci *Register* a přidá vložení potřebných hlavičkových souborů. Jednotku můžeme uložit a dát ji smysluplné jméno.
13. Chování komponenty můžeme testovat při běhu programu před její instalací na Paletu komponent. To je užitečné pro ladění nově vytvářených komponent, ale můžeme tuto techniku použít pro testování libovolných komponent, a to bez ohledu na to, zda komponenta je již zobrazena na Paletě komponent. Testování neinstalovaných komponent děláme emulací akcí prováděných Builderem, když uživatel umístí komponentu z palety na formulář. Provedeme tyto činnosti:
    - Vytvoříme novou aplikaci nebo otevřeme existující.
    - Vložíme hlavičkový soubor jednotky komponenty do hlavičkového souboru jednotky formuláře.
    - Přidáme položku objektu reprezentujícího komponentu k typu formuláře. To je hlavní rozdíl mezi způsobem přidávání komponenty a způsobem prováděným Builderem. Přidáme položku do veřejné části na závěr deklarace typu formuláře. Builder ji přidává výše, do části deklarace, která ji zpracovává. My ji

tam nemůžeme přidat. Prvky v této části deklarace typu odpovídají prvkům uloženým v souboru formuláře. Přidání jména komponenty, která neexistuje na formuláři, vytvoří chybný soubor formuláře.

- V konstruktoru formuláře vytvoříme komponentu. Když voláme konstruktor komponenty, musíme předat parametr specifikující vlastníka komponenty. Je vhodné jako vlastníka předávat vždy **this**. V metodě je **this** odkaz na objekt obsahující metodu.
- Nastavíme vlastnost **Parent**. Nastavení vlastnosti **Parent** je vždy první věcí provedenou po vytvoření ovladače. **Parent** určuje komponentu, která vizuálně obsahuje ovladač (často to bývá formulář, ale může to být i panel nebo **GroupBox**). Normálně **Parent** nastavujeme na **this**, tj. na formulář. **Parent** vždy nastavujeme před nastavením ostatních vlastností ovladače. Jestliže komponenta není ovladačem (tj. jestliže některým z jejím předků není **TControl**), přeskočíme tento krok (nastavení vlastnosti **Parent** v tomto případě způsobí chybu).
- Nastavíme podle potřeby další vlastnosti komponenty.

Předpokládejme, že chceme testovat novou komponentu **TNovaKomponenta** uloženou v jednotce pojmenované *NovyCtrl*. Vytvoříme nový projekt a provedeme výše popsané kroky. Dostaneme tuto jednotku formuláře (nejdříve je vypsán jeho hlavičkový soubor).

```
#ifndef Unit1H
#define Unit1H
#include <vcl\Classes.hpp>
#include <vcl\Controls.hpp>
#include <vcl\StdCtrls.hpp>
#include <vcl\Forms.hpp>
#include "NovyCtrl.h"   // 2. Přidáme NovyCtrl do hlavičkového souboru formuláře
class TForm1 : public TForm
{
__published:      // IDE-managed Components
private:   // User declarations
public:           // User declarations
   TNovaKomponenta* NovyOvladac;                    // 3. Přidáme datovou položku
     __fastcall TForm1(TComponent* Owner);
};
extern TForm1 *Form1;
#endif

#include <vcl\vcl.h>
#pragma hdrstop
#include "Unit1.h"
#pragma resource "*.dfm"
TForm1 *Form1;
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
  NovyOvladac = new TNovaKomponenta(this);  // 4. Konstruktor komponenty
  NovyOvladac->Parent=this; // 5. Nastavení vlastnosti Parent, je-li to ovladač
  NovyOvladac->Left=12;     // 6. Nastavení ostatních vlastností podle potřeby
}
```

14. Když instalujeme komponentu na Paletu komponent, pak je knihovna komponent přebudována. Kdykoli je knihovna komponent přebudovávána (při instalaci komponenty nebo po volbě **Component | Rebuild Library**), Builder vytváří zdrojový soubor knihovny. Jméno zdrojového souboru knihovny je jméno knihovny s příponou CPP. Pro VCL jméno souboru je CMPLIB32.CPP. K uložení zdrojového kódu generované knihovny zvolíme **Options | Environment | Library | Save Library Source Code**. Před instalací naší nové komponenty přesuneme všechny soubory komponenty do adresáře *Builder\Lib\Obj*. Jedná se o soubory: všechny binární soubory (DFM, RES nebo RC a DCR), všechny zdrojové soubory (CPP a PAS), všechny soubory OBJ a LIB a všechny hlavičkové soubory (H a HPP). Do knihovny komponent můžeme přidávat komponenty zapsané v C++ nebo Pascalu. K přidání komponenty do knihovny komponent zvolíme **Component | Install** a v zobrazeném dialogovém okně stiskneme **Add** k otevření dialogového okna **Add Module**. Zde zapíšeme jméno jednotky, kterou chceme přidat nebo zvolíme **Browse** a jednotku najdeme. Stiskem **OK** dialogové okno **Add Module** uzavřeme. Jména jednotek komponent, které jsme specifikovali jsou uvedeny na konci seznamu *Installed Components*. Jestliže v tomto seznamu vybereme jméno některé jednotky, pak jména tříd komponent umístěných v této jednotce jsou zobrazeny v seznamu *Component classes*. Jména tříd pro nově přidávané komponenty nejsou zobrazena. Stiskem **OK** uzavřeme dialogové okno **Install Components** a přebudujeme knihovnu. Nově instalované komponenty jsou přidány na Paletu komponent a můžeme je používat ve svých aplikacích. Paletu komponent můžeme modifikovat volbou **Component | Configure Palette**.

15. Práce s Builderem využívá myšlenku, že objekt obsahuje data i kód a že s objektem můžeme manipulovat jak během návrhu, tak i při běhu aplikace. V tomto smyslu vnímá komponenty jejich uživatel. Při vytváření nových komponent, pracujeme s objekty způsobem, který koncový uživatel nikdy nepotřebuje. Před zahájením tvorby komponent, je nutno se dobře seznámit s principy objektově orientovaného programování popsanými dále. Základní rozdíl mezi uživatelem komponent a tvůrcem komponent je ten, že uživatel manipuluje s instancemi objektů a tvůrce vytváří nové typy objektů.

    Účelem definování tříd komponent je poskytnout základ pro užitečné instance. Tj. cílem je vytvořit objekt, který my nebo jiní uživatelé mohou používat v různých aplikacích, v různých situacích nebo alespoň v různých částech stejné aplikace. Jsou dva důvody k odvození nové třídy: změna implicitního typu, kterou se vyhneme opakování a přidání nových možností k typu. V obou případech, je cílem vytvoření opakovaně použitelných objektů. Ve všech programovacích úlohách se snažíme vyhnout se opakování. Jestliže potřebujeme několikrát zapsat stejné řádky kódu, pak je můžeme umístit do funkce nebo dokonce vytvořit knihovnu funkcí, kterou můžeme používat v mnoha programech. Stejný důvod je i pro komponenty. Jestliže často měníme stejné vlastnosti nebo provádíme stejné volání metod, je užitečné vytvořit nový typ komponenty, který tyto věci provede implicitně. Např. předpokládejme, že pokaždé při vytváření aplikace, chceme přidat formulář dialogového okna k provedení jisté funkce. Ačkoliv není obtížné pokaždé znova vytvořit dialogové okno, není to ale nutné. Můžeme navrhnout okno pouze jednou, nastavit jeho vlastnosti a výsledek instalovat na Paletu komponent jako znovupoužitelnou komponentu. Toto nejen redukuje opakování, ale také provádí standardizaci. Jiným důvodem pro vytváření nového typu komponenty je přidání možností, které zatím existující komponenta nemá. Lze to provést odvozením od existujícího typu komponenty (např. vytvoření specializovaného typu seznamu) nebo od abstraktního základního typu, jako je **TComponent** nebo **TControl**. Novou komponentu vždy odvozujeme od typu, který obsahuje největší podmnožinu požadovaných služeb. Objektu můžeme přidávat nové vlastnosti, ale nemůžeme je odebírat, tj. jestliže existující typ komponenty obsahuje vlastnosti, které nechceme vložit do své komponenty, musíme komponentu odvodit od předka komponenty. Např. jestliže chceme přidat nějaké možnosti k seznamu, můžeme odvodit novou komponentu od **TListBox**. Nicméně, jestliže chceme přidat nějaké nové možnosti, ale odstranit některé existující možnosti standardního seznamu, musíme odvodit svůj nový seznam od **TCustomListBox**, tj. předka **TListBox**. Znovuvytvoříme možnosti seznamu, které chceme použít a přidáme své nové možnosti.

    Když se rozhodneme, že je nutno odvodit nový typ komponenty, musíme také určit od kterého typu komponenty svou komponentu odvodíme (viz výše). Builder poskytuje několik abstraktních typů komponent určených pro tvůrce komponent k odvozování nových typů komponent. K deklarování nového typu komponenty, přidáme deklaraci typu do hlavičkového souboru jednotky komponenty. Následující příklad je deklarace jednoduché grafické komponenty:

```
class TPrikladTvaru : public TGraphicControl
{
public:
  virtual __fastcall TPrikladTvaru(TComponent *Owner);
};
```

    K dokončení deklarace komponenty vložíme do třídy deklarace vlastností, položek a metod, ale prázdná deklarace je také přípustná a poskytuje počáteční bod pro vytváření komponenty.

16. Pro uživatele komponent je komponenta entitou obsahující vlastnosti, metody a události. Uživatel nemusí znát co z toho komponenta zdědila a od koho to zdědila. Toto je ale značně důležité pro tvůrce komponenty. Uživatel komponenty si může být jist, že každá komponenta má vlastnosti **Top** a **Left**, které určují, kde bude komponenta zobrazena na formuláři, který ji vlastní. Nemusí znát, že všechny komponenty dědí tyto vlastnosti od společného předka **TComponent**. Nicméně, když vytváříme komponenty, musíme znát, který objekt dědí, od kterého objektu příslušnou část. Musíme také znát co naše komponenta dědí a můžeme tak využít zděděné služby bez jejich znovuvytvoření. Z definice třídy vidíme, že když odvozujeme třídu, odvozujeme ji od existující třídy. Třída od které odvozujeme se nazývá bezprostřední předek naší nové třídy. Předek bezprostředního předka je také předek nové třídy; jsou to všechno předkové. Nová třída je potomek svých předků. Jestliže nespecifikujeme předka třídy, Builder odvozuje třídu od implicitního předka **TObject**. Standardní typ **TObject** je předkem všech tříd v Knihovně vizuálních komponent.

    Všechny vztahy předek-potomek v aplikaci tvoří hierarchii tříd. Nejdůležitější k zapamatování v hierarchii tříd je to, že každá generace tříd obsahuje více než její předkové. Tj. třída dědí vše co obsahuje předek a přidává nová data a metody nebo předefinovává existující metody. Nicméně, třída nemůže odstranit nic z toho co zdědila. Např. jestliže třída má jistou vlastnost, pak všechny přímí nebo nepřímí potomci mají také tuto vlastnost.

17. C++ poskytuje pět úrovně řízení přístupu k částem tříd. Řízení přístupu určuje, který kód může přistupovat ke které části třídy. Specifikací úrovní přístupu, definujeme rozhraní naší komponenty. Pokud nespecifiku-

jeme jinak, pak položky, metody a vlastnosti přidané do naší třídy jsou soukromé. Následující tabulka ukazuje úrovně přístupu v pořadí od nejvíce omezujícího k nejméně omezujícímu:

Úroveň	Používá se pro
private	Skrytí implementačních detailů
protected	Definování rozhraní vývojáře
public	Definování rozhraní pro běh programu
__published	Definování rozhraní pro návrh
__automated	Pro automatizaci OLE

Deklarací části třídy jako soukromé, uděláme tuto část neviditelnou z kódu mimo třídu, pokud funkce není přítelem třídy. Soukromé části tříd jsou užitečné pro ukrytí implementačních detailů před uživateli komponent. Jelikož uživatelé objektu nemohou přistupovat k soukromé části, můžeme změnit vnitřní implementaci objektu bez vlivu na kód uživatele.

Deklarací části třídy jako chráněné, uděláme tuto část neviditelnou z kódu mimo třídu, což je stejné jako u soukromé části. Rozdíl u chráněné části je ten, že třída odvozená od tohoto typu, může přistupovat k jejím chráněným částem. Chráněné deklarace můžeme použít k definování rozhraní návrháře objektu. Tj. uživatel objektu nemá přístup k chráněným částem, ale vývojář (např. tvůrce komponent) ano. Můžeme tedy udělat rozhraní přístupným tak, že tvůrci komponent je mohou v odvozených třídách měnit, s tím, že tyto detaily nejsou viditelné pro koncové uživatele.

Deklarací části třídy jako veřejné, uděláme tuto část viditelnou pro jakýkoli kód, který má přístup ke třídě jako celku. Tj. veřejné části nemají žádné omezení. Veřejné části třídy jsou dostupné za běhu programu pro všechen kód a veřejné části třídy definují v tomto objektu rozhraní běhu programu. Rozhraní běhu programu je užitečné pro prvky, které nezpracováváme v době návrhu, jako jsou vlastnosti, které závisí na aktuálních informacích o typech za běhu programu nebo které jsou určeny pouze pro čtení. Metody, které slouží pro uživatele našich komponent také deklarujeme jako část rozhraní běhu programu. Poznamenejme, že vlastnosti určené pouze pro čtení nemůžeme používat během návrhu a uvádíme je ve veřejné části deklarace.

Deklarací částí třídy jako zveřejňované, uděláme tuto část veřejnou, která také generuje informace o typech za běhu programu pro tuto část. Mimo jiné informace o typech za běhu programu zajišťují, že Inspektor objektů může přistupovat k vlastnostem a událostem. Protože pouze zveřejňovaná část je zobrazována v Inspektoru objektů, zveřejňovaná část třídy určuje rozhraní objektu pro návrh. Rozhraní objektu pro návrh zahrnuje všechny aspekty objektu, které uživatel objektu může chtít přizpůsobit během návrhu, ale nesmí obsahovat vlastnosti, které závisí na informacích o prostředí běhu programu.

18. Vyřízení metod je termín použitý k popisu, jak naše aplikace určuje, který kód bude proveden při volání metody. Když zapisujeme kód, který volá metodu objektu, je to stejné, jako volání jiné funkce. Nicméně objekty mají dva různé způsoby vyřízení metod. Tyto dva typy vyřízení metod jsou: statické a virtuální. Virtuální metody se ale podstatně liší od statických metod. Typy vyřízení metod jsou důležité pro pochopení, jak vytvářet komponenty.

    Všechny metody jsou statické, pokud nespecifikujeme v jejich deklaraci něco jiného. Statické metody pracují jako volání normálních funkcí. Překladač určí adresu metody a připojí metodu během překladu. Základní výhodou statických metod je, že jejich vyřízení je velmi rychlé. Protože překladač může určit adresu metody, metoda je volána přímo. Virtuální metody používají nepřímé hledání adresy jejich metod při běhu programu, což je mnohem delší. Další rozdíl u statické metody je ten, že se nemění v odvozených typech. Tj. když deklarujeme třídu, která obsahuje statickou metodu, potom odvozením nové třídy, potomek třídy sdílí přesně stejnou metodu na stejné adrese. Statické metody tedy vždy provádějí to samé, bez ohledu na aktuální typ objektu. Statickou metodu nelze předefinovat. Deklarováním metody v typu potomka se stejným jménem jako má statická metoda v objektu předka se nahradí metoda předka. Např. v následujícím kódu první komponenta deklaruje dvě statické metody. Druhá deklaruje dvě statické metody se stejným jménem, které nahradí obě metody v první komponentě.

```
class TPrvniKomponenta : public TComponent
{
  void Presun();
  void Zablesk();
}
class TDruhaKomponenta : public TPrvniKomponenta
{
  void Presun();
  void Zablesk();
}
```

    Volání virtuálních metod je stejné jako volání jiných metod, ale mechanismus jejich vyřízení je složitější. Virtuální metody umožňují předefinování v objektech potomků, ale stále metodu voláme stejným způsobem.

Adresa volané metody není určena při překladu, ale je hledána až při běhu aplikace. K deklaraci nové virtuální metody, přidáme direktivu **virtual** před deklaraci metody. Direktiva **virtual** v deklaraci metody vytváří položku v tabulce virtuálních metod (VTM) objektu. VTM obsahuje adresy všech virtuálních metod v typu objektu. Když odvozujeme novou třídu od existující třídy, nová třída získá svou vlastní VTM, která obsahuje všechny položky z VTM svého předka a položky dalších virtuálních metod deklarovaných v novém objektu. Potomek třídy může předefinovat některé ze zděděných virtuálních metod. Předefinování metody znamená její rozšíření nebo změnu, namísto jejího nahrazení. Třída potomka může opětovně deklarovat a implementovat libovolnou z metod deklarovaných ve svých předcích. Statickou metodu nelze předefinovat, neboť deklarace statické metody se stejným jménem jako má zděděná statická metoda, nahradí zděděnou metodu kompletně. K předefinování metody z třídy předka, předeklarujeme metodu v odvozené třídě s tím, že počet a typy parametrů se musí shodovat. Následující kód ukazuje deklaraci dvou jednoduchých komponent. První deklaruje dvě metody, každá je jiného typu vyřízení. Druhá odvozená od první, nahrazuje statickou metodu a předefinovává virtuální metodu.

```
class TPrvniKomponenta : public TComponent
{
  void Presun();
  virtual void Zablesk();
}
class TDruhaKomponenta : public TPrvniKomponenta
{
  void Presun();
  void Zablesk();
}
```

19. Jednou z věcí, kterou si musíme uvědomit, když vytváříme komponentu je to, že použití existující komponenty je realizováno ukazatelem. Toto se stává důležité, když předáváme objekty jako parametry. Při předávání objektů je vhodnější použít parametr volaný hodnotou než odkazem. Objekty jsou ve skutečnosti ukazateli, které jsou již odkazem. Předáním objektu odkazem je vlastně předáván odkaz na odkaz.
20. Vlastnosti jsou nejdůležitější částí komponent, neboť uživatelé komponent je mohou vidět a pracovat s nimi během návrhu a získat tak bezprostřední zpětnou vazbu na reakci komponenty v reálném čase. Vlastnosti jsou také důležité, neboť usnadňují používání komponent. Vlastnosti poskytují významné výhody a to jak pro tvůrce komponent, tak i pro jejich uživatele. Nejzřejmější výhodou je, že vlastnost může být v době návrhu zobrazena v Inspektoru objektů. To zjednodušuje naše programování, neboť namísto zadávání několika parametrů při vytváření objektu, zpracujeme hodnoty přiřazené uživatelem. Pro uživatele komponent se vlastnosti podobají proměnným. Uživatel může nastavovat nebo číst hodnoty vlastností, jako kdyby tyto vlastnosti byly položkami objektů. Rozdíl je pouze v tom, že vlastnost nemůže být použita jako parametr volaný odkazem. Vlastnosti poskytují více možností než položky objektu neboť: **Uživatel může nastavovat vlastnosti během návrhu.** To je velmi důležité, neboť narozdíl od metod, které jsou dostupné pouze při běhu aplikace, vlastnosti slouží k přizpůsobení komponenty před spuštěním aplikace. Komponenta nemusí obsahovat mnoho metod, které by zapouzdřovaly tyto vlastnosti. **Narozdíl od položek objektu, vlastnosti mohou skrýt implementační detaily před uživateli.** Např. data mohou být uložena v zakódovaném tvaru, ale když nastavujeme nebo čteme hodnotu vlastnosti potřebujeme je nezakódované. Přestože hodnota vlastnosti může být číslo, komponenta může hledat hodnotu v databázi nebo k získání hodnoty provádět složité výpočty. **Vlastnosti poskytují k příkazu přiřazení vedlejší efekt.** Když provedeme přiřazení, je volána metoda, která může dělat cokoliv. Příkladem je vlastnost **Top** všech komponent. Přiřazení nové hodnoty vlastnosti **Top**, neznamená jenom změnu nějaké uložené hodnoty, ale způsobí i přemístění a překreslení komponenty samotné. Efekt nastavení vlastnosti není omezen na nějakou komponentu. Nastavení vlastnosti **Down** komponenty urychlovacího tlačítka na *true*, způsobí nastavení **Down** všech ostatních urychlovacích tlačítek ve skupině na *false*. **Implementace metody pro vlastnost může být virtuální**, což znamená, že může provádět různé věci v různých komponentách.
21. Vlastnost může být libovolného typu. Důležitým aspektem volby typu pro naši vlastnost je to, že různé typy jsou různě zobrazovány v Inspektoru objektů. Inspektor objektů používá typ vlastnosti k určení co uživatel chce zobrazit. Při registraci komponenty můžeme specifikovat různé editory vlastností. V následující tabulce je uvedeno jak vlastnost je zobrazena v Inspektoru objektů.

Typ vlastnosti	Zacházení s vlastností v Inspektoru objektů
Jednoduchý	Číselné, znakové a řetězcové vlastnosti se zobrazují v Inspektoru objektů jako čísla, znaky nebo řetězce. Uživatel může zadávat a editovat hodnotu vlastnosti přímo.
Výčtový	Vlastnosti výčtových typů (včetně **bool**) zobrazují hodnotu tak, jak je definována ve zdrojovém kódu. Uživatel může cyklicky procházet možnými hodnotami dvo-

jitým kliknutím ve sloupci hodnot. Hodnotu můžeme také vybírat ze seznamu obsahujícího všechny možné hodnoty výčtového typu.

Množina — Vlastnosti typu množina se zobrazují v Inspektoru objektů jako množiny. Rozšířením množiny, uživatel může zacházet s každým prvkem množiny jako s logickou hodnotou: *true*, jestliže prvek je obsažen v množině nebo *false* není-li.

Objekt — Vlastnost, která je sama objektem, často má svůj vlastní editor vlastností. Nicméně, jestliže objekt, který je vlastností má také zveřejňované vlastnosti, pak Inspektor objektů umožňuje uživateli rozšířit seznam vlastností objektu a editovat je samostatně. Objektové vlastnosti musí být odvozeny od **TPersistent**.

Pole — Pole vlastností musí mít svůj vlastní editor vlastností. Inspektor objektů nemá zabudovánu podporu pro editaci pole vlastností.

22. Všechny komponenty dědí vlastnosti od svých předků. Když odvozujeme novou komponentu od existujícího typu komponenty, naše nová komponenta dědí všechny vlastnosti ze třídy předka. Jestliže odvozujeme komponentu od jednoho z abstraktních typů, pak zděděné vlastnosti jsou chráněné nebo veřejné, ale ne zveřejňované. Aby chráněná nebo veřejná vlastnost byla přístupná pro uživatele komponenty, musíme ji opětovně deklarovat jako zveřejňovanou. To provedeme přidáním deklarace zděděné vlastnosti do deklarace třídy potomka. Jestliže odvozujeme komponentu od **TWinControl**, komponenta např. zdědí vlastnost **Ctl3D**, ale tato vlastnost je chráněná a uživatel komponenty nemůže k **Ctl3D** během návrhu ani při běhu programu. Opětovnou deklarací **Ctl3D** v naši nové komponentě, můžeme změnit úroveň ochrany na veřejnou nebo zveřejňovanou. Následující kód ukazuje opětnou deklaraci **Ctl3D** jako zveřejňovanou, což ji zpřístupní během návrhu:

```
class TPrikladKomponenty : public TWinControl
{
__published
  __property Ctl3d;
}
```

Opětovnou deklarací můžeme pouze zmírnit omezení přístupu a nelze je zvětšit. Chráněnou vlastnost lze změnit na veřejnou, ale nelze změnit veřejnou vlastnost na chráněnou. Při opakované deklaraci, specifikujeme pouze jméno vlastnosti, typ a další informace se neuvádí. Můžeme také deklarovat novou implicitní hodnotu a ukládací specifikátory.

23. Deklarace vlastnosti a její implementace je snadná. Přidáme deklaraci vlastnosti k deklaraci třídy naší komponenty. V deklaraci vlastnosti specifikujeme tři věci: jméno vlastnosti, typ vlastnosti a metody pro čtení a nastavování hodnot vlastnosti. Vlastnosti komponent minimálně musíme deklarovat ve veřejné části deklarace typu objektu komponenty, což umožní číst a nastavovat vlastnosti z vnějšku komponenty při běhu programu. K vytvoření editovatelných komponent během návrhu musíme deklarovat vlastnost ve zveřejňované části deklarace třídy komponenty. Zveřejňované vlastnosti jsou automaticky zobrazovány v Inspektoru objektů. Veřejné vlastnosti jsou přístupné pouze za běhu programu. Následuje typická deklarace vlastnosti:

```
class TNaseKomponenta : public TComponent
{
private:
  int FPocet;                              // položka pro uložení vlastnosti
  int __fastcall GetPocet();                    // čtecí metoda
  void __fastcall SetPocet(int APocet);         // zápisová metoda
public:
  __property int Pocet={read=GetPocet,write=SetPocet}    //deklarace vlastnosti
}
```

Není žádné omezení jak ukládat data vlastnosti. Komponenty Builderu používají ale tyto konvence: Data vlastnosti jsou ukládány v položkách třídy. Identifikátor pro položku vlastnosti třídy začíná písmenem **F** a následuje jméno vlastnosti. Např. data pro vlastnost **Width** definovanou v **TControl** jsou uložena v položce objektu nazvané **FWidth**. Položku objektu pro vlastnost deklarujeme jako soukromou. To zajistí, že komponenta, která deklaruje vlastnost, má k ní přístup, ale uživatelé komponenty a potomci komponenty ne. Potomci komponenty mohou používat samotnou zděděnou vlastnost, ale nemají přístup k vnitřnímu uložení dat komponenty. Základním principem těchto konvencí je, že pouze implementace přístupových metod vlastnosti mohou přistupovat k datům této vlastnosti. Jestliže metoda nebo jiná vlastnost potřebuje změnit tato data, musí to provést prostřednictvím vlastnosti a ne přímým přístupem k uloženým datům. To zajišťuje, že implementaci zděděné metody můžeme měnit bez vlivu na potomky komponenty.

24. Nejjednodušším způsobem zpřístupnění dat je přímý přístup. Tj. části **read** a **write** deklarace vlastnosti specifikují, že přiřazení nebo čtení hodnoty vlastnosti probíhá přímo s položkou vnitřního uložení bez volání přístupové metody. Přímý přístup je užitečný, když vlastnost nemá vedlejší efekty, ale chceme ji zpřístupnit v Inspektoru objektů. Často používáme přímý přístup pro část **read** deklarace vlastnosti a přístupovou

metodu pro část **write** neboť obvykle aktualizujeme stav komponenty na základě nové hodnoty vlastnosti. Následující deklarace typu komponenty ukazuje vlastnost, která používá přímý přístup pro čtení i zápis:

```
class TPrikladKomponenty : public TComponent
{
private:
  bool FPouzeProCteni;      //vnitřní uložení dat je soukromé
__published:                //umožňuje přístup k vlastnosti během návrhu
  __property bool PouzeProCteni={read=FPouzeProCteni,write=FPouzeProCteni};
}
```

25. Syntaxe deklarace vlastnosti umožňuje, aby části **read** a **write** deklarace vlastnosti specifikovaly přístupové metody namísto položek objektu. Bez ohledu na implementaci částí **read** a **write** jisté vlastnosti, tato implementace musí být soukromá a potomci komponent mohou k zděděné vlastnosti přistupovat. To zajistí, že použití vlastnosti není ovlivněno změnami v implementaci. Udělání přístupových metod soukromými také zajistí, že uživatelé komponent nemohou tyto metody volat k modifikaci vlastnosti. Čtecí metoda pro vlastnost je funkce, která nemá parametry a vrací hodnotu stejného typu jako má vlastnost. Podle konvencí, jméno funkce začíná **Get** a pokračuje jménem vlastnosti. Např. čtecí metoda pro vlastnost nazvanou **Pocet** by se měla jmenovat **GetPocet**. Výjimkou k „funkce je bez parametrů“ je případ pole parametrů, které předává jejich indexy jako parametry. Čtecí metoda pracuje s vnitřním uložením dat a vytváří hodnotu vlastnosti příslušného typu. Je-li vlastnost typu „pouze pro zápis“, není nutné deklarovat čtecí metodu. Vlastnosti, určené pouze pro zápis se používají velmi zřídka a obecně nejsou moc užitečné. Zápisová metoda pro vlastnost je vždy funkce typu **void** s jedním parametrem, který je stejného typu jako vlastnost. Parametr může být předáván odkazem nebo hodnotou a jeho jméno může být libovolné. Podle konvencí jméno funkce je **Set** následované jménem vlastnosti. Např. zápisová metoda pro vlastnost nazvanou **Pocet** by se měla jmenovat **SetPocet**. Hodnota předaná v parametru je použita k nastavení nové hodnoty vlastnosti a zápisová metoda musí provést operace potřebné k vytvoření příslušné hodnoty ve vnitřním formátu. Je-li vlastnost typu pouze pro čtení, není nutné deklarovat zápisovou metodu. Je vhodné testovat, zda se nová hodnota liší od současné hodnoty před jejím přiřazením. Např. následuje příklad zápisové metody pro vlastnost typu *int* nazvanou **Pocet**, která ukládá svou aktuální hodnotu v položce **FPocet**:

```
void __fastcall TMojeKomponenta::SetPocet(int Hodnota)
{
  if (Hodnota != FPocet) {
    FPocet = Hodnota;
    Update();
  }
}
```

26. Když deklarujeme vlastnost, můžeme pro ní volitelně deklarovat implicitní hodnotu. Implicitní hodnota vlastnosti komponenty je hodnota nastavená pro tuto vlastnost konstruktorem komponenty. Např. když umístíme komponentu z Palety komponent na formulář, Builder vytváří komponentu voláním konstruktoru komponenty, který určuje počáteční hodnoty vlastností komponenty. Builder používá deklarované implicitní hodnoty k určení, zda ukládat vlastnost v souboru formuláře. Jestliže implicitní hodnotu pro vlastnost nespecifikujeme, pak Builder vlastnost vždy ukládá. K deklarování implicitní hodnoty pro vlastnost připojíme direktivu **default** k deklaraci (nebo opětovné deklaraci) následovanou implicitní hodnotou. Deklarací implicitní hodnoty v deklaraci vlastnosti nenastavujeme aktuálně vlastnost na tuto hodnotu. Jako tvůrce komponenty musíme zajistit, že konstruktor komponenty nastaví vlastnost na tuto hodnotu. Když opakovaně deklarujeme vlastnost, můžeme specifikovat, že vlastnost nemá implicitní hodnotu, i když zděděná vlastnost ji má. K určení, že vlastnost nemá implicitní hodnotu, připojíme direktivu **nodefault** k deklaraci vlastnosti. Když deklarujeme vlastnost poprvé, není nutno specifikovat **nodefault**, protože absence deklarace implicitní hodnoty znamená totéž. Následuje deklarace komponenty, která obsahuje vlastnost **IsTrue** typu *bool* s implicitní hodnotou *true* a konstruktor nastavující implicitní hodnotu:

```
class TPrikladKomponenty : public TComponent
{
private:
  bool FIsTrue;
public:
  virtual __fastcall TPrikladKomponenty(TComponent* Owner);
__published:
  __property bool IsTrue = {read=FIsTrue, write=FIsTrue, default=true};
};
__fastcall TPrikladKomponenty::TPrikladKomponenty(TComponent * Owner)
  : TComponent(Owner)
{
```

```
  FIsTrue = true;                    //nastavení implicitní hodnoty
}
```

Pokud by implicitní hodnota pro **IsTrue** měla být *false*, potom ji není nutno explicitně nastavovat v konstruktoru, neboť všechny třídy (a tedy i komponenty) vždy inicializují všechny své položky na nulu a „nulová“ logická hodnota je *false*.

27. Přejděme ke konkrétnímu příkladu. Budeme modifikovat standardní komponentu **Memo** k vytvoření komponenty, která implicitně neprovádí lámání slov a má žluté pozadí. Je to velmi jednoduchý příklad, ale ukazuje vše, co je zapotřebí provést k modifikaci existující komponenty. Implicitně hodnota vlastnosti **WordWrap** komponenty **Memo** je *true*. Jestliže používáme několik těchto komponent u nichž nechceme provádět lámání slov (automatický přechod na další řádek při dosažení konce řádku) můžeme snadno vytvořit novou komponentu, která implicitně lámání slov neprovádí (implicitní hodnotu vlastnosti **WordWrap** nastavíme na *false*). Modifikace existující komponenty probíhá ve dvou krocích: vytvoření a registrace komponenty a modifikace objektu komponenty. Základní proces je ale vždy stejný, ale u složitějších komponent budeme muset provést více kroků pro přizpůsobení nového objektu. Vytváření každé komponenty začíná stejně: vytvoříme programovou jednotku, odvodíme třídu komponenty, registrujeme ji a instalujeme ji na Paletu komponent. V našem příkladě použijeme uvedený obecný postup s těmito specifikami: jednotku komponenty nazveme *Memos*, od **TMemo** odvodíme nový typ komponenty nazvaný **TWrapMemo** a registrujeme **TWrapMemo** na stránce *Samples* Palety komponent. Výsledek naši práce je tento (nejdříve je uveden výpis hlavičkového souboru):

```
#ifndef MemosH
#define MemosH
#include <vcl\SysUtils.hpp>
#include <vcl\Controls.hpp>
#include <vcl\Classes.hpp>
#include <vcl\Forms.hpp>
#include <vcl\StdCtrls.hpp>
class TWrapMemo : public TMemo
{
private:
protected:
public:
__published:
};
#endif

#include <vcl\vcl.h>
#pragma hdrstop
#include "Memos.h"
namespace Memos
{
  void __fastcall Register()
  {
    TComponentClass classes[1] = {__classid(TWrapMemo)};
    RegisterComponents("Samples", classes, 0);
  }
}
```

V tomto případě jsme nepoužili k vytvoření komponenty Experta komponent, ale vytvořili jsme ji manuálně. Pokud bychom použili Experta komponent, pak by byl do vytvořené třídy automaticky přidán konstruktor.

Všechny komponenty nastavují své hodnoty vlastností při vytváření. Když umístíme komponentu během návrhu na formulář nebo když spustíme aplikaci vytvářející komponentu a přečteme její vlastnosti ze souboru formuláře, je nejprve volán konstruktor komponenty k nastavení implicitních hodnot komponenty. V případě zavedení komponenty ze souboru formuláře, po vytvoření objektu s jeho implicitními hodnotami vlastností, aplikace dále nastavuje vlastnosti změněné při návrhu a když je komponenta zobrazena vidíme ji tak, jak jsme ji navrhli. Konstruktor ale vždy určuje implicitní hodnoty vlastností. Pro změnu implicitní hodnoty vlastnosti, předefinujeme konstruktor komponenty k nastavení určené hodnoty. Když předefinujeme konstruktor, pak nový konstruktor musí vždy volat zděděný konstruktor a to dříve než provedeme cokoliv jiného. V našem příkladě, naše nová komponenta musí předefinovat konstruktor zděděný od **TMemo** k nastavení vlastnosti **WordWrap** na *false* a vlastnosti **Color** na *clYellow*. Přidáme tedy deklaraci předefinovaného konstruktoru do deklarace třídy a zapíšeme nový konstruktor do CPP souboru:

```
class TWrapMemo : public TMemo
{
```

```
public:
    __fastcall TWrapMemo(TComponent* Owner);
};

__fastcall TWrapMemo::TWrapMemo(TComponent* Owner)
    : TMemo(Owner)
{
  Color = clYellow;
  WordWrap = false;
}
```

Nyní můžeme instalovat novou komponentu na Paletu komponent a přidat ji na formulář. Vlastnost **WordWrap** je nyní implicitně *false* a vlastnost **Color** *clYellow*. Jestliže změníme (nebo vytvoříme) novou implicitní hodnotu vlastnosti, musíme také určit, že hodnota je implicitní. Jestliže to neprovedeme, Builder nemůže ukládat a obnovovat hodnotu vlastnosti. Když Builder ukládá popis formuláře do souboru formuláře, ukládá pouze hodnoty vlastností, které se liší od jejich implicitních hodnot. Má to dvě výhody: zmenšuje to soubor formuláře a urychluje zavádění formuláře. Jestliže vytvoříme vlastnost nebo změníme implicitní hodnotu vlastnosti je vhodné aktualizovat deklaraci vlastnosti na novou implicitní hodnotu. Ke změně implicitní hodnoty vlastnosti, opětovně deklarujeme vlastnost a připojíme direktivu **default** s novou implicitní hodnotou. Není nutno opětovně deklarovat prvky vlastnosti, pouze jméno a implicitní hodnotu. V našem příkladě provedeme:

```
class TWrapMemo : public TMemo
{
public:
    __fastcall TWrapMemo(TComponent* Owner);
__published:
  __property Color = {default=clYellow};
  __property WordWrap = {default=false};
};
```

Specifikace implicitní hodnoty vlastnosti nemá vliv na celkovou práci komponenty. Musíme stále explicitně nastavit implicitní hodnotu v konstruktoru komponenty. Rozdíl je ve vnitřní práci aplikace: Builder nezapisuje **WordWrap** do souboru formuláře, jestliže je *false*, neboť předpokládá, že konstruktor nastaví tuto hodnotu automaticky. Totéž platí i o vlastnosti **Color**.

28. Vytvořte komponentu **FontCombo** (kombinované okno volby písma). Tuto komponentu odvoďte od komponenty **ComboBox** tak, že změníte implicitní hodnotu vlastnosti **Style** na *csDropDownList* a do vlastnosti **Items** přiřadíte *Screen->Fonts*. Vyzkoušejte použití této komponenty v nějaké aplikaci (bude signalizována chyba).
29. Namísto přiřazení seznamu písem v konstruktoru komponenty **FontCombo** je nutno jej přiřadit v metodě *CreateWnd* (inicializuje parametry), kterou předefinujeme (nejprve voláme metodu předka a potom provedeme přiřazení). Vyzkoušejte provést tuto změnu. Nyní již tuto komponentu můžeme používat bez problémů.
30. V dalším zadání se pokusíme modifikovat komponentu **ListBox** a to tak, aby v zobrazeném textu mohly být použity znaky tabulátorů (ke stylu okna je nutno přidat příznak *LBS_USETABSTOPS*). Komponentu nazveme **TabList** a předefinujeme v ní metodu *CreateParams*, kterou Builder používá ke změně některých standardních hodnot používaných při tvorbě komponenty. Tato metoda bude vypadat takto:

```
void __fastcall TTabBox::CreateParams(Controls::TCreateParams &Params)
{
  TListBox::CreateParams(Params);
  Params.Style |= LBS_USETABSTOPS;
}
```

Komponentu vyzkoušejte.

14. Další informace o komponentách

1. Některé vlastnosti mohou být indexované, podobně jako pole. Mají více hodnot, které rozlišujeme indexem. Příkladem ve standardních komponentách je vlastnost **Lines** komponenty **Memo**. **Lines** je indexovaný seznam řetězců, které tvoří text komponenty a můžeme k nim přistupovat jako k poli řetězců. V tomto případě, pole vlastností dává uživateli přirozený přístup k jistému prvku (řetězci - řádku) ve větší množině dat (textu komponenty). Pole vlastností pracuje stejně jako ostatní vlastnosti a deklarujeme je většinou stejně (jsou zde pouze tyto rozdíly): Deklarace vlastnosti obsahuje jeden nebo více indexů určitého typu. Indexy mohou být libovolného typu. Části **read** a **write** deklarace vlastností, jsou-li specifikovány, pak musí být metodami. Nelze zde specifikovat položky třídy. Přístupové metody pro čtení a zápis hodnoty vlastnosti přibírají další

parametry, které odpovídají indexu nebo indexům. Parametry musí být ve stejném pořadí a stejného typu jako indexy specifikované v deklaraci vlastnosti. Na rozdíl od indexu pole, typ indexu pro pole vlastností nemusí být celočíselného typu. Např. jako index pole vlastností může být i řetězec. Můžeme se také odkazovat na individuální prvky pole vlastností, neležící v rozsahu vlastnosti. Následuje deklarace vlastnosti, která vrací na základě celočíselného indexu řetězec:

```
Class TPrikladKomponenty : public TComponent
{
private:
  System::AnsiString __fastcall GetJmenoCisla(int Index);
public:
  __property System::AnsiString JmenoCisla[int Index] = {read=GetJmenoCisla};
};

System::AnsiString __fastcall TPrikladKomponenty::GetJmenoCisla(int Index)
{
  System::AnsiString Vysledek;
  switch (Index){
    case 0:
      Vysledek = "Nula";
      break;
    case 100:
      Vysledek = "Malé";
      break;
    case 1000:
      Vysledek = "Velké";
      break;
    default Vysledek = "Neznámé";
  }
  return Vysledek;
}
```

2. Inspektor objektů poskytuje možnost editace pro všechny typy vlastností. Nicméně můžeme poskytnout alternativní editor pro konkrétní vlastnost zápisem a registrací editoru vlastnosti. Můžeme registrovat editor vlastnosti, který lze použít pouze na vlastnosti ve vytvářené komponentě, ale můžeme také vytvořit editor, který lze použít na všechny vlastnosti jistého typu. V nejjednodušší úrovni editor vlastností může operovat jedním nebo oběma z těchto způsobů: zobrazovat a zpřístupnit k editování uživateli současnou hodnotu jako textový řetězec a zobrazit dialogové okno provádějící některé typy editace. V závislosti na editované vlastnosti je užitečné poskytnout jeden nebo oba způsoby. Zápis editoru vlastností vyžaduje pět kroků: odvození třídy editoru vlastností, editace vlastnosti jako text, editace vlastnosti jako celek, specifikaci atributů editoru a registrace editoru vlastností.

   Soubor DSGNINTF.HPP definuje několik typů editoru vlastností, všechny jsou odvozeny od **TPropetryEditor**. Když vytváříme editor vlastností, můžeme třídu editoru vlastností odvodit přímo od **TPropertyEditor** nebo nepřímo od jednoho z typů editoru vlastností popsaných v další tabulce (odvozujeme nový objekt od jednoho z existujícího typu editoru vlastností). Soubor DSGNINTF.HPP také definuje některé velmi specializované editory vlastností používané pro unikátní vlastnosti jako je např. jméno komponenty.

Typ	Edituje vlastnosti
TOrdinalProperty	Základ pro všechny editory vlastností ordinálních typů (celočíselné, znakové a výčtové vlastnosti).
TIntegerProperty	Všechny celočíselné typy včetně předdefinovaných a uživatelem definovaných intervalů.
TCharPropetry	Typ Char a intervaly Char, jako 'A'..'Z'.
TEnumProperty	Libovolný výčtový typ.
TFloatProperty	Libovolné reálné číslo.
TStringPropetry	Řetězce.
TSetElementProperty	Individuální prvek v množině, zobrazovaný jako logická hodnota.
TSetPropetry	Všechny množiny. Množiny nejsou přímo editovatelné, ale můžeme ji rozšířit na seznam prvků množiny.
TClassPropetry	Objekty. Zobrazí jméno typu objektu a umožňuje expandovat vlastnosti objektu.
TMethodProperty	Ukazatelé metod.
TComponentProperty	Komponenty na formuláři. Uživatel nemůže editovat vlastností komponent, ale může ukazovat na specifické komponenty kompatibilního typu.

TColorPropetry	Komponenta barev. Roletový seznam obsahuje konstanty barev. Dvojité kliknutí otevírá dialogové okno výběru barvy.
TFontNamePropetry	Jména písma. Roletový seznam obsahuje všechny aktuálně instalovaná písma.
TFontProperty	Písma. Umožňuje rozšířit na individuální vlastnosti písma stejně jako přístup k dialogovému oknu písma.

Jedním z jednoduchých editorů vlastností je **TFloatProperty**, editor pro vlastnosti, které jsou reálnými čísly. Následuje jeho deklarace:

```
class TFloatProperty : public TPropertyEditor
{
  typedef TPropertyEditor inherited;
public:
  virtual bool __fastcall AllEqual(void);
  virtual System::AnsiString __fastcall GetValue(void);
  virtual void __fastcall SetValue(const System::AnsiString Value);
};
```

3. Všechny vlastnosti musí poskytovat řetězcovou reprezentaci svých hodnot pro zobrazení v Inspektoru objektů. Mnoho vlastností také zpřístupňuje uživateli typ v nové hodnotě pro vlastnost. Objekt editoru vlastností poskytuje virtuální metody, které můžeme předefinovat pro převod mezi textovou reprezentací a aktuální hodnotou. Tyto metody jsou nazvané **GetValue** a **SetValue**. Náš editor vlastností také dědí množinu metod používaných pro přiřazení a čtení jiných typů hodnot (viz následující tabulka):

Typ vlastnosti	Metoda „Get“	Metoda „Set“
reálný	GetFloatValue	SetFloatValue
ukazatel metody (události)	GetMethodValue	SetMethodValue
ordinální typ	GetOrdValue	SetOrdValue
řetězec	GetStrValue	SetStrValue

Když předefinujeme metodu **GetValue**, můžeme stále volat jednu z metod „Get“ a když předefinujeme metodu **SetValue**, můžeme stále volat jednu z metod „Set“. Metoda **GetValue** editoru vlastností vrací řetězec, který reprezentuje současnou hodnotu vlastnosti. Inspektor objektů používá tento řetězec ve sloupci hodnot pro vlastnost. Implicitně **GetValue** vrací `'unknown'`. K poskytnutí řetězcové reprezentace pro naši vlastnost, předefinujeme metodu **GetValue** editoru vlastností. Jestliže vlastnost nemá řetězcovou hodnotu, naše **GetValue** musí převést hodnotu na její řetězcovou reprezentaci. Metoda **SetValue** editoru vlastností přebírá řetězec zapsaný uživatelem v Inspektoru objektů, převádí jej na příslušný typ a nastavuje hodnotu vlastnosti. Jestliže řetězec nemá reprezentaci příslušné hodnoty pro vlastnost, **SetValue** generuje výjimku a nevhodnou hodnotu nepoužije. Pro přečtení řetězcové hodnoty z vlastnosti, předefinujeme metodu **SetValue** editoru vlastnosti.

4. Volitelně můžeme poskytnout dialogové okno, ve kterém může uživatel viditelně editovat vlastnost. Toto je užitečné pro editory vlastností jejichž vlastnosti jsou sami třídy. Příkladem je vlastnost **Font**, pro kterou uživatel může otevřít dialogové okno k volbě všech atributů písma. K poskytnutí dialogového okna celkového editoru vlastností, předefinujeme metodu **Edit** třídy editoru vlastností. Metoda **Edit** používá stejné metody „Get“ a „Set“ jako jsou použity v metodách **GetValue** a **SetValue** (metoda **Edit** volá obě tyto metody). Jelikož editor je specifického typu, obvykle není potřeba převádět hodnotu vlastnosti na řetězec. Když uživatel klikne na tlačítko ‘...’ vedle vlastnosti nebo dvojitě klikne ve sloupci hodnot, Inspektor objektů volá metodu **Edit** editoru vlastnosti. V implementaci metody **Edit** provedeme tyto kroky: Vytvoříme náš editor, přečteme současné hodnoty a přiřadíme je metodou „Get“, když uživatel změní některou hodnotu, přiřadíme tuto hodnotu pomocí metody „Set“ a zrušíme editor.
5. Editor vlastností musí poskytovat informace, které Inspektor objektů může použít k určení zobrazeného nástroje. Např. Inspektor objektů musí znát, zda vlastnost má podvlastnosti nebo zda může zobrazit seznam možných hodnot. Pro specifikaci atributů editoru, předefinujeme metodu **GetAttributes** objektu editoru. **GetAttributes** je metoda vracející množinu hodnot typu **TPropertyAttributes**, která může obsahovat některé nebo všechny následující hodnoty:

Příznak	Význam, je-li vložen	Ovlivňuje metodu
paValueList	Editor může zobrazit seznam výčtových hodnot.	GetValues
paSubProperties	Vlastnost má podvlastnosti, které může zobrazit.	GetProperties
paDialog	Editor může pro editaci zobrazit dialogové okno.	Edit
paMultiSelect	Uživatel může vybrat více než jeden prvek.	
paAutoUpdate	Aktualizuje komponentu po každé změně, namísto čekání na schválení hodnoty.	SetValue
paSortList	Inspektor objektů seřadí seznam hodnot.	

paReadOnly	Uživatel nemůže modifikovat hodnotu vlastnosti.
paRevertable	Povoluje volbu **Revert to Inherited** v místní nabídce Inspektora objektů (návrat k předchozí hodnotě).

Vlastnost **Color** má několik možností, jak ji uživatel zvolí v Inspektoru objektů: zápisem, výběrem ze seznamu a editorem. Metoda **GetAttributes TColorProperty**, tedy obsahuje několik atributů ve své návratové hodnotě:

```
Virtual __fastcall TPropertyAttributes TColorProperty::GetAttributes()
{
  return TPropertyAttributes()<<paMultiSelect<<paDialog<<paValueList;
}
```

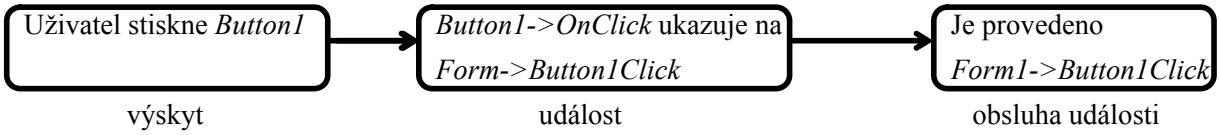
6. Vytvořený editor vlastností se musí v Builderu registrovat. Registrací editoru vlastností přiřadíme typ vlastnosti k editoru vlastnosti. Můžeme registrovat editor se všemi vlastnostmi daného typu nebo s jistou vlastností jistého typu komponenty. K registraci editoru vlastností voláme metodu **RegisterPropetryEditor**. Tato metoda má čtyři parametry: Prvním je ukazatel na informace o typu pro editovanou vlastnost. To je vždy volání funkce **__typeinfo**, např. `__typeinfo(TMojeKomponenta)`. Druhým je typ komponenty, na který je tento editor aplikován. Jestliže tento parametr je NULL, editor je použitelný na všechny vlastnosti daného typu. Třetím parametrem je jméno vlastnosti. Tento parametr má význam pouze, jestliže předchozí parametr specifikuje konkrétní typ komponenty. V tomto případě, můžeme specifikovat jméno konkrétní vlastnosti v typu komponenty, se kterou tento editor pracuje. Posledním parametrem je typ editoru vlastnosti použitý pro editování specifikované vlastnosti. Následuje ukázka funkce, která registruje editory pro standardní komponenty na Paletě komponent:

```
namespace Newcomp
{
  void __fastcall Register()
  {
    RegisterPropertyEditor(__typeinfo(TComponent), 0L, "",
                           __classid(TComponentProperty));
    RegisterPropertyEditor(__typeinfo(TComponentName), TComponent,
                          "Name", __classid(TComponentNameProperty));
    RegisterPropertyEditor(__typeinfo(TMenuItem), TMenu, "",
                           __classid(TMenuItemProperty));
  }
}
```

Tři příkazy v této funkci ukazují různé použití **RegisterPropertyEditor**: První příkaz je nejtypičtější. Registruje editor vlastností **TComponentProperty** pro všechny vlastnosti typu **TComponent** (nebo potomků **TComponent**, které nemají registrován vlastní editor). Obecně, když registrujeme editor vlastností, vytvoříme editor pro jistý typ a chceme jej použít pro všechny vlastnosti tohoto typu, pak jako druhý parametr použijeme NULL a jako třetí parametr prázdný řetězec. Druhý příkaz je nejspecifičtějším typem registrace. Registruje editor pro jistou vlastnost v jistém typu komponenty. V tomto případě je to editor pro vlastnost **Name** všech komponent. Třetí příklad je specifičtější než první, ale není limitován jako druhý. Registruje editor pro všechny vlastnosti typu **TMenuItem** v komponentách typu **TMenu**.

7. **Události** jsou velmi důležitou částí komponent. Jsou propojením mezi výskytem v systému (jako je např. akce uživatele nebo změna zaostření), na který komponenta může reagovat a částí kódu, který reaguje na tento výskyt. Reagující kód je obsluha události a je většinou zapisována uživatelem komponenty. Pomocí událostí, vývojář aplikace může přizpůsobit chování komponenty a to bez nutnosti změny samotného objektu. Jako tvůrce komponenty, použijeme události k povolení vývojáři aplikace přizpůsobit chování komponenty. Události pro mnoho akcí uživatele (např. akce myši) jsou zabudovány ve všech standardních komponentách Builderu, ale můžeme také definovat nové události. Builder implementuje události jako vlastnosti.

   Obecně řečeno, událost je mechanismus, který propojuje výskyt s určitým kódem. Více specificky, událost je závěr (ukazatel), který ukazuje na určitou metodu v určité instanci třídy. Z perspektivy uživatele komponenty, událost je jméno svázané s událostí systému, jako je např. **OnClick**, kterému uživatel může přiřadit volání určité metody. Např. stisknutí tlačítka nazvaného **Button1** má metodu **OnClick**. Implicitně Builder generuje obsluhu události nazvanou **Button1Click** ve formuláři, který obsahuje tlačítko a přiřadí ji **OnClick**. Když se vyskytne událost kliknutí na tlačítku, tlačítko volá metodu přiřazenou **OnClick**, v tomto případě **Button1Click**.



Uživatel komponenty tedy používá událost ke specifikaci uživatelského kódu, který aplikace volá při výskytu jisté události.

8. Builder používá k implementaci událostí závěry. Závěr je speciální typ ukazatele, který ukazuje na určitou metodu v určité instanci objektu. Jako tvůrce komponenty můžeme používat závěr jako adresu místa. Náš kód detekuje výskyt události a je volána metoda (je-li) specifikovaná uživatelem pro tuto událost. Závěr obhospodařuje skrytý ukazatel na instanci třídy. Když uživatel přiřadí obsluhu k události komponenty, nepřiřadí metodu jistého jména, ale jistou metodu jisté instance objektu. Tato instance je obvykle formulář obsahující komponentu, ale nemusí jim být. Všechny ovladače např. dědí virtuální metodu nazvanou **Click** pro zpracování události kliknutí:
```
virtual void __fastcall Click(void);
```
Jestliže uživatel má přiřazenou obsluhu k události **OnClick** ovladače, pak výskytem kliknutí na ovladači je volání přiřazené obsluhy. Jestliže obsluha není přiřazena, neprovádí se nic.

9. Komponenty používají k implementaci svých událostí vlastnosti. Narozdíl od jiných vlastností, události nemohou použít metody k implementování částí **read** a **write**. Je zde nutno použít soukromou položku objektu a to stejného typu jako je vlastnost. Podle konvencí, jméno položky je stejné jako jméno vlastnosti, ale na začátek je přidáno písmeno **F**. Např. ukazatel metody **OnClick** je uložen v položce nazvané **FOnClick** typu **TNotifyEvent** a deklarace vlastnosti události **OnClick** je tato:
```
class TControl : public TComponent
{
private:
  TNotifyEvent FOnClick;      {deklarace položky k uložení ukazatele metody}
  ...
protected:
  __property TNotifyEvent OnClick = {read=FOnClick, write=FOnClick};
...
};
```
Stejně jako u jiných vlastností, můžeme nastavovat nebo měnit hodnotu události při běhu aplikace a pomocí Inspektora objektů může uživatel komponent přiřazovat obsluhu při návrhu.

10. Protože událost je ukazatel na obsluhu události, typ vlastnosti události musí být závěrem. Podobně, libovolný kód použitý jako obsluha události musí být odpovídajícím typem metody třídy. Všechny metody obsluh událostí jsou funkce typu **void**. Jsou kompatibilní s událostí daného typu, metoda obsluhy událostí musí mít stejný počet a typy parametrů a musí být předány ve stejném pořadí. Builder definuje typy metod pro všechny své standardní události. Když vytváříme svou vlastní událost, můžeme použít existující typ (pokud vyhovuje) nebo definovat svůj vlastní. Přestože obsluha události je funkce, nesmíme hodnotu funkce nikdy použít při zpracování události (funkce musí být typu **void**). Prázdná funkce vrací nedefinovaný výsledek, prázdná obsluha události, která by vracela hodnotu, by byla chybná. Jelikož obsluha událostí nemůže vracet hodnotu, musíme získávat informace zpět z uživatelova kódu prostřednictvím parametrů volaných odkazem. Příkladem předávání parametrů volaných odkazem obsluze události je událost stisku klávesy, která je typu **TKeyPressEvent**. **TKeyPressEvent** definuje dva parametry, první, indikující, který objekt generuje událost a druhý indikující, která klávesa byla stisknuta:
```
typedef void __fastcall (__closure *TKeyPressEvent)(TObject *Sender,Char &Key);
```
Normálně, parametr *Key* obsahuje znak stisknutý uživatelem. V některých situacích může uživatel komponenty chtít tento znak změnit. Např. může chtít převést znaky malých písmen na odpovídající písmena velká. V tomto případě může uživatel definovat následující obsluhu události pro klávesnici:
```
void __fastcall TForm1::Edit1KeyPress(TObject *Sender,Char &Key)
{
  Key = UpCase(Key);
}
```
Při vytváření událostí komponenty musíme pamatovat na to, že uživatel našich komponent nemusí připojit obsluhu k události. To znamená, že naše komponenta negeneruje chybu, pokud uživatel komponenty nepřipojí obsluhu k jisté události.

11. Všechny ovladače v Builderu dědí události pro nejdůležitější události Windows. Tyto události nazýváme standardní události. Všechny tyto události zabudované v abstraktních ovladačích, jsou implicitně chráněné, což znamená, že koncový uživatel k nim nemůže připojit obsluhu. Když vytvoříme ovladač, můžeme zvolit, zda bude událost viditelná pro uživatele našeho ovladače. Jsou dvě kategorie standardních událostí: události definované pro všechny ovladače a události definované pouze pro standardní okenní ovladače. Nejzákladnější události jsou definovány v typu objektu **TControl**. Všechny ovladače (okenní, grafické nebo uživatelské) dědí tyto události. Následuje seznam událostí přístupných ve všech ovladačích:

OnClick OnDragDrop OnEndDrag OnMouseMove OnDblClick OnDragOver

OnMouseDown OnMouseUp

Všechny standardní události mají chráněné virtuální metody deklarované v **TControl**, jejichž jméno odpovídá jménu události, ale je bez „On“ na začátku. Např. událost **OnClick** volá metodu jména **Click**. Mimo události společné pro všechny ovladače, mají ovladače odvozené od **TWinControl** další události (mají také příslušné metody):

OnEnter OnKeyDown OnKeyPress OnKeyUp OnExit

Deklarace standardních události jsou chráněné a chráněné jsou i metody, které jim odpovídají. Jestliže chceme tyto vlastnosti zpřístupnit uživateli při běhu programu nebo při návrhu, musíme opětovně deklarovat vlastnost události jako veřejnou nebo zveřejňovanou. Opětovná deklarace vlastnosti bez specifikace její implementace zachovává implementovanou metodu, ale změní úroveň ochrany. Tím můžeme zviditelnit standardní události. Jestliže vytváříme komponentu, např. chceme zpřístupnit událost **OnClick** během návrhu, přidáme do deklarace typu komponenty:

```
class TMujOvladac : public TCustomControl
{
__published:
  __property OnClick;    {zviditelní OnClick v Inspektoru objektů}
};
```

12. Jestliže chceme změnit způsob reakce komponenty na jistou třídu událostí, zapíšeme příslušný kód a přiřadíme jej události. Pro uživatele komponenty to je přesně to co chce. Nicméně, když vytváříme komponentu, není to co chceme, protože musíme udržet událost přístupnou pro uživatele komponenty. Je to smysl chráněné implementace metod přiřazených ke každé standardní události. Předefinováním implementace metody, můžeme modifikovat vnitřní obsluhu události a voláním zděděné metody můžeme obsloužit standardní zpracování, včetně uživatelova kódu. Je důležité kdy voláme zděděnou metodu. Obecné pravidlo je volat zděděnou metodu nejdříve, a použít kód původní obsluhy události dříve než svůj přizpůsobený. Nicméně, někdy chceme provést svůj kód před voláním zděděné metody. Např. jestliže zděděný kód je nějak závislý na stavu komponenty a náš kód mění tento stav, pak chceme změnit stav a nechat uživatelův kód reagovat na změněný stav. Předpokládejme např. že zapisujeme komponentu a chceme modifikovat způsob reakce nové komponenty na kliknutí. Namísto přiřazení obsluhy k události **OnClick**, jak by to provedl uživatel komponenty, předefinujeme chráněnou metodu **Click**:

```
void __fastcall TMujOvladac::Click() {
  TWinControl::Click();
  // naše přizpůsobení vložíme sem
}
```

13. Definování nových událostí je relativně vzácné. Mnohem častěji provádíme předefinování již existujících událostí. Nicméně někdy, když chování komponenty je značně odlišné, pak je vhodné definovat pro ni událost. Definování události probíhá ve čtyřech krocích: spuštění události, definování typu obsluhy, deklarace události a volání události. První co provedeme, když definujeme svou vlastní událost, která neodpovídá žádné standardní události, je určení co událost spustí. Pro některé události je odpověď zřejmá. Např. když uživatel stiskne levé tlačítko myši, Windows zasílá zprávu WM_LBUTTONDOWN aplikaci. Po příjmu této zprávy komponenta volá svou metodu **MouseDown**, která dále volá nějaký kód, který uživatel má připojen k události **OnMouseDown**. Ale u některých událostí je obtížnější určit, co specifikuje externí událost. Např. posuvník má událost **OnChange**, spouštěnou několika typy výskytů, včetně klávesnice, kliknutí myši nebo změnou v jiném ovladači. Když definujeme svou událost, musíme zajistit, že všechny možné výskyty spustí naši událost.

    Jsou dva typy výskytů, pro které musíme události ošetřit: změna stavu a akce uživatele. Mechanismus zpracování je stejný, ale liší se sémanticky. Událost akce uživatele je téměř vždy spouštěna zprávou Windows, indikující, že uživatel provedl něco na co naše komponenta má reagovat. Událost změny stavu je také svázána se zprávou od Windows (např. změna zaostření nebo povolení něčeho), ale může také vzniknou na základě změny vlastnosti nebo jiného kódu. Musíme definovat, že to vše může spustit událost. Když určíme jak naše událost vznikne, musíme definovat jak ji chceme obsloužit. To znamená určit typ obsluhy události. V mnoha případech, obsluhy pro události definujeme sami jednoduchým oznámením nebo typem specifickým události. To také umožňuje získat informace zpět z obsluhy. Oznamovací událost je událost, která pouze říká, se jistá událost nastala a nespecifikuje žádné informace o ní. Oznámení používá typ **TNotifyEvent**, který má pouze jeden parametr a je to odesilatel události. Tedy všechny obsluhy pro oznámení znají o události pouze to, o jakou třídu události se jedná a která komponenta událost způsobila. Např. události kliknutí jsou oznámení. Když zapisujeme obsluhu pro událost kliknutí, vše co známe je to, že kliknutí nastalo a na které komponentě bylo kliknuto. Oznámení jsou jednosměrný proces. Není mechanismus k poskytnutí zpětné vazby nebo zabránění budoucí obsluhy oznámení. V některých případech nám ale tyto informace nestačí. Např. jestliže událost je událost stisku klávesy, je pravděpodobné, že chceme také znát, která klávesa

byla stisknuta. V těchto případech požadujeme typ obsluhy, který obsahuje parametry s nějakými nezbytnými informacemi o události. Jestliže naše událost je generována v reakci na zprávu, je pravděpodobné, že parametry předávané obsluze události získáme přímo z parametrů zprávy. Protože všechny obsluhy událostí jsou funkce vracející **void**, jediný způsob k předání informací zpět z obsluhy je pomocí parametru volaného odkazem. Naše komponenta může použít tyto informace k určení jak a co událost provede po provedení obsluhy uživatele. Např. všechny události klávesnice (**OnKeyDown**, **OnKeyUp** a **OnKeyPress**) předávají hodnotu stisknuté klávesy v parametru volaném odkazem jména *Key*. Obsluha události může změnit *Key* a tak aplikace vidí jinou klávesu než která způsobila událost. To je např. způsob jak změnit zapsaný znak na velká písmena.

Když jsme určili typ naší obsluhy události, můžeme deklarovat ukazatel metody a vlastnosti pro událost. Události dáme smysluplné a popisné jméno tak, aby uživatel pochopil, co událost dělá. Je vhodné, aby bylo konzistentní s jmény podobných vlastností v jiných komponentách. Jména všech standardních událostí v Builderu začínají „**On**“. Je to pouze konvence, překladač nevyžaduje její dodržování. Inspektor objektů určuje že vlastnost je událost podle typu vlastnosti: všechny vlastnosti ukazatelů metod jsou považovány za události a jsou zobrazeny na stránce událostí.

Obecně je vhodné centralizovat volání události, tj. vytvořit virtuální metodu v naší komponentě, která volá uživatelskou obsluhu události (pokud ji uživatel přiřadí) a provádí nějaké implicitní zpracování. Umístění všech volání událostí na jednom místě zajistí, že nějaká odvozená nová komponenta od naší komponenty může přizpůsobit obsloužení události předefinováním jen jedné metody, namísto hledáním v našem kódu místa, kde událost je volána. Nesmíme nikdy vytvořit situaci ve které prázdná obsluha události způsobí chybu, tj. vlastní funkčnost našich komponent nesmí záviset na jisté reakci z kódu uživatelovi obsluhy události. Z tohoto důvodu, prázdná obsluha musí produkovat stejný výsledek jako neobsloužená. Komponenty nikdy nesmějí požadovat aby uživatel je použil jistým způsobem. Důležitým aspektem je princip, že uživatel komponent nemá žádné omezení na to, co může s nimi dělat v obsluze události. Jelikož prázdná obsluha se má chovat stejně jako žádná obsluha, kód pro volání uživatelské obsluhy má vypadat takto:

```
if (OnClick) OnClick(this);
// provedení implicitního zpracování
```

Nikdy nesmíme použít tento způsob:

```
if (OnClick) OnClick(this);
else
// provedení implicitního zpracování
```

Pro některé typy událostí, uživatel může chtít nahradit implicitní zpracování. K umožnění aby to mohl udělat, musíme přidat parametr volaný odkazem k obsluze a testovat jej na jistou hodnotu při návratu z obsluhy. Když např. zpracováváme událost stisku klávesy, uživatel může požadovat implicitní zpracováni nastavením parametru **Key** na nulový znak (viz následující příklad):

```
if (OnKeyPress) OnKeyPress(this, &Key);
if (Key != NULL)            // provedení implicitního zpracování
```

Skutečný kód se nepatrně liší od zde uvedeného (spolupracuje se zprávou Windows), ale logika je stejná. Implicitně komponenta volá nějakou uživatelem přiřazenou obsluhu a pak provede své standardní zpracování. Jestliže uživatelova obsluha nastaví **Key** na nulový znak komponenta přeskočí implicitní zpracování.

14. Metody komponent se neliší od ostatních metod objektu. Jsou to funkce zabudované ve struktuře třídy komponenty. Obecným pravidlem je minimalizovat počet metod, které uživatel naší komponenty může volat. Lépe než metody je využívat vlastnosti. Vždy, když zapisujeme komponentu minimalizujeme podmínky vkládané na uživatele komponenty. Do nejvyšší míry by měl být uživatel komponenty schopný dělat cokoli s komponentou a kdykoli to chce. Jsou situace, kdy toto nelze splnit, ale naším cílem je nejvíce se k tomu přiblížit. Přestože je nemožné vypsat všechny druhy závislostí, kterým se chceme vyhnout, následující seznam nabízí různé věci, kterým se máme vyhýbat: Metodám, které uživatel musí volat, aby mohl používat komponentu, metodám, které musí být použity v určitém pořadí a metodám, které způsobí, že určité události nebo metody mohou být chybné. Např. když vyvolání metody způsobí, že naše komponenta přejde do stavu, kdy volání jiné metody může být chybné, pak napíšeme tuto jinou metodu tak, že když ji uživatel zavolá a komponenta je ve špatném stavu, pak metoda opraví tento stav před provedením vlastního kódu. Minimálně bychom měli zajistit generování výjimky v případech, kdy uživatel použije nedovolenou metodu. Jinými slovy, jestliže vytvoříme situaci, kde části našeho kódu jsou vzájemně závislé, musíme zajistit, aby používání kódu chybným způsobem nezpůsobilo uživateli problémy. Varování je lepší než havárie systému, když se uživatel nepřizpůsobí našim vzájemným závislostem.

   Builder neklade žádná omezení na jména metod a jejich parametrů. Nicméně je několik konvencí, které usnadňují používání metod pro uživatele naších komponent. Je vhodné dodržovat tato doporučení: Volíme popisná jména. Jméno jako **PasteFromClipboard** je mnohem více informativní než jednoduché **Paste** nebo **PFC**. Ve jménech svých funkcí používáme slovesa. Např. **ReadFileNames** je mnohem srozumitelnější než

DoFiles. Jména funkcí by měla vyjadřovat, co vracejí. Přestože funkci vracející vodorovnou souřadnici něčeho můžeme nazvat *X*, je mnohem výhodnější použít jméno **GetHorizontalPosition**. Ujistěte se, že metody skutečně musí být metodami. Dobrá pomůcka je, že jméno metody obsahuje sloveso. Jestliže vytvoříme několik metod, jejich jméno neobsahuje sloveso, zamyslete se nad tím, zda by tyto metody nemohly být vlastnostmi.

15. Všechny části objektů, včetně položek, metod a vlastností mohou mít různé úrovně ochrany. Volba vhodné úrovně ochrany pro metody je velmi důležitá. Všechny metody, které uživatelé našich komponent mohou volat musí být deklarované jako veřejné. Musíme mít na paměti, že mnoho metod je voláno v obsluhách událostí, a tak metody by se měli vyhnout plýtváním systémovými zdroji nebo uvedením Windows do stavu, kdy nemůže reagovat. Konstruktory a destruktory musí být vždy veřejné.

    Metody, které jsou implementačními metodami pro komponentu musí být chráněné. To zabrání uživateli v jejich volání v nesprávný čas. Jestliže máme metody, které uživatelský kód nesmí volat, ale objekty potomků volat mohou, pak metody deklarujeme jako chráněné. Např. předpokládejme, že máme metodu která spoléhá na nastavení jistých dat předem. Jestliže tuto metodu uděláme veřejnou, umožníme tím uživateli, aby ji volat i před nastavením potřebných dat. Pokud ale bude chráněná, zajistíme tím, že uživatel ji nemůže volat přímo. Můžeme pak vytvořit jinou veřejnou metodu, která zajistí nastavení dat před voláním chráněné metody.

    Jedinou kategorií metod, které musí být vždy soukromé jsou implementační metody vlastností. Tím znemožníme uživateli ve volání metod, které manipulují s daty vlastnosti. Zajišťují, že jediný přístup k těmto informacím je prostřednictvím samotné vlastnosti. Jestliže objekty potomků požadují předefinování metod implementace, mohou to udělat, ale namísto předefinování implementačních metod, musí zpřístupnit zděděnou hodnotu vlastnosti prostřednictvím samotné vlastnosti.

    Virtuální metody v komponentách Builderu se neliší od virtuálních metod v jiných třídách. Metodu uděláme virtuální, když chceme aby různé typy byly schopny provést různý kód v reakci na stejné funkční volání. Jestliže vytváříme komponentu určenou pouze k přímému použití koncovým uživatelem, uděláme pravděpodobně všechny její metody statické. Na druhé straně, jestliže vytváříme komponentu více abstraktní povahy, kterou jiní tvůrci komponent mohou použít jako počáteční bod pro své vlastní komponenty, zvážíme vytvoření virtuálních metod. Tímto způsobem, komponenty odvozené od naši komponenty mohou předefinovat zděděné virtuální metody.

    Deklarování metod v komponentách se neliší od deklarování metod v jiných třídách. Při deklaraci nové metody v komponentě provedeme dvě věci: přidáme její deklaraci k deklaraci třídy komponenty a implementujeme metodu v souboru CPP jednotky komponenty. Následující kód ukazuje komponentu, která definuje dvě nové metody: jednu chráněnou statickou metodu a jednu veřejnou virtuální metodu.

```
class TPrikladKomponenty : public TControl
{
protected:
  void __fastcall Zvetsi();
public:
  virtual int __fastcall VypoctiOblast();
};
...
void __fastcall TPrikladKomponenty::Zvetsi()
{
  Height = Height + 5;
  Width = Width + 5;
}
int __fastcall TPrikladKomponenty::VypoctiOblast()
{
  return Width * Height;
}
```

16. Windows poskytuje účinné GDI (Graphics Device Interface) pro kreslení grafiky nezávislé na zařízení. Naneštěstí GDI má velmi mnoho požadavků na programátora, jako je např. správa grafických zdrojů, což má za následek, že strávíme mnoho času prováděním jiných věcí než tím co skutečně chceme dělat, tj. tvořit grafiku. Builder se za nás stará o GDI, což dovoluje strávit více času produktivní prací a nemusíme hledat ztracená madla nebo neuvolněné zdroje. Builder se stará o vedlejší úkoly za nás a tak se můžeme zaměřit na produktivní a tvořivou činnost. Funkce GDI můžeme také volat přímo z aplikací Builderu. Builder ale také tyto grafické funkce zaobaluje a umožňuje tak pracovat při vytváření grafiky produktivněji. Builder obaluje GDI Windows na několika úrovních. Nejdůležitější pro nás jako tvůrce komponent je způsob zobrazení obrazu komponenty na obrazovce. Když voláme funkce GDI přímo, musíme mít madlo kontextu zařízení, ve kterém můžeme vybírat různé kreslící nástroje jako jsou pera, štětce a písma. Po dokreslení našeho grafického

obrazu, musíme obnovit kontext zařízení do jeho původního stavu. Namísto používání této nízké grafické úrovně, Builder poskytuje jednoduché ale kompletní rozhraní: vlastnost **Canvas** naší komponenty. Plátno přebírá zjišťování zda má přípustný kontext zařízení, a uvolňuje kontext, když již není používán. Plátno má své vlastní vlastnosti reprezentující aktuální pero, štětec a písmo. Plátno obhospodařuje všechny tyto zdroje za nás, a není nutno se tedy zabývat vytvářením, výběrem a uvolňováním věcí jako je madlo pera. Stačí říci plátnu, který typ pera chceme použít a plátno zajistí zbytek. Jednou z výhod obhospodařování grafických zdrojů v Builderu je to, že může uschovat zdroje pro pozdější použití, což může značně urychlit opakování operací. Např. jestliže máme program, který opakovaně vytváří, použije a uvolní jistý typ nástroje pera, není stále nutné opakovat tyto kroky. Protože Builder uchovává grafické zdroje, opakovaně použije již existující nástroj pera. Jako příklad, jak jednoduchý grafický kód Builderu může být, následují dvě ukázky kódu. První používá standardní funkce GDI k nakreslení žluté elipsy modře orámované na okně v aplikaci zapsané v *ObjectWindows*. Druhá používá plátno k nakreslení stejné elipsy v aplikaci zapsané v Builderu.

```
void TMojeOkno::Paint(TDC& PaintDC, bool erase, TRect& rect)
{
  HPEN PenHandle, OldPenHandle;
  HBRUSH BrushHandle, OldBrushHandle;
  PenHandle=CreatePen(PS_SOLID, 1, RGB(0,0,255)); {vytvoření modrého pera}
  OldPenHandle=SelectObject(PaintDc,PenHandle); {říká DC aby používal modré pero}
  BrushHandle=CreateSolidBrush(RGB(255,255,0)); {vytvoření žlutého štětce}
  OldBrushHandle=SelectObject(PaintDC,BrushHandle); {říká DC aby jej používal}
  Ellipse(HDC, 10, 10, 50, 50);          {nakreslení elipsy}
  SelectObject(OldBrushHandle);          {obnovení původního štětce}
  DeleteObject(BrushHandle);             {zrušení žlutého štětce}
  SelectObject(OldPenHandle);            {obnovení původního pera}
  DeleteObject(PenHandle);               {zrušení modrého pera}
}

void __fastcall TForm1::FormPaint(TObject *Sender)
{
  Canvas->Pen->Color = clBlue;                {udělání modrého pera}
  Canvas->Brush->Color = clYellow;            {udělání žlutého štětce}
  Canvas->Ellipse(10, 10, 50, 50);            {nakreslení elipsy}
}
```

Objekt plátna obaluje grafiku Windows na několika úrovních. Vysoká úroveň obsahuje funkce pro kreslení čar, tvarů a textů, prostřední úroveň pro manipulaci s kreslícími možnostmi plátna a nízká úroveň pro přístup k GDI Windows. Tyto možnosti jsou uvedeny v následující tabulce:

Úroveň	Operace	Nástroje
Vysoká	Kreslení čar a tvarů	Metody jako MoveTo, LineTo, Rectangle a Ellipse
	Zobrazení a umístění textu	Metody TextOut, TextHight, TextWidth a TextRect
	Vyplňování oblastí	Metody FillRect a FloodFill
Střední	Přizpůsobení textu a grafiky	Vlastnosti Pen, Brush a Font
	Manipulace s body	Vlastnost Pixels
	Kopírování a spojování obrázků	Metody Draw, StretchDraw, BrushCopy a CopyRect a vlastnost CopyMode
Nízká	Volání funkcí GDI Windows	Vlastnost Handle

Podrobnější informace o objektu plátna a jeho metodách a vlastnostech získáme v nápovědě.

17. Mnoho práce v Builderu je omezeno na kreslení přímo na plátno komponent a formulářů. Builder také umožňuje zpracovávat standardní obrázky jako jsou bitové mapy, metasoubory a ikony, včetně automatické správy palet. V Builderu jsou tři typy objektů určených pro práci s grafikou: **Plátna**, která reprezentují bitově mapovanou kreslící plochu na formuláři, grafickém ovladači, tiskárně nebo bitové mapě. Plátno je vždy vlastnost něčeho, ale nikdy to není standardní objekt. **Grafika**, která reprezentuje grafické obrazy uložené v souborech nebo zdrojích, jako jsou bitové mapy, ikony nebo metasoubory. Builder definuje typy objektů **TBitmap**, **TIcon** a **TMetafile**, všechny odvozené od generického **TGraphic**. Můžeme také definovat své vlastní grafické objekty. Definováním minimálního standardního rozhraní pro všechnu grafiku, **TGraphic** poskytuje jednoduchý mechanismus pro aplikace k snadnému použití různých typů grafiky. **Obrázky**, které jsou kontejnery pro grafiku, mohou obsahovat typy libovolných grafických objektů. Tj. prvek typu **TPicture** může obsahovat bitovou mapu, ikonu, metasoubor nebo uživatelem definovaný grafický typ a aplikace k nim může přistupovat stejným způsobem prostřednictvím objektu obrázku. Např. ovladač **Image** má vlastnost nazvanou **Picture**, typu **TPicture** umožňující řízení zobrazování obrázků z mnoha typů grafiky. Objekt obrázku má vždy grafiku a grafika má plátno (plátno má pouze **TBitmap**). Normálně, když pracujeme s obrázkem, pak pracujeme pouze s částí grafického objektu přístupného prostřednictvím **TPicture**.

Jestliže požadujeme přístup ke specifikám samotného grafického objektu, můžeme se odkázat na vlastnost **Graphic** obrázku.

Všechny obrázky a grafika v Builderu mohou zavádět svůj obraz ze souboru a ukládat jej opět zpět (nebo do jiných souborů). Můžeme zavádět a ukládat obraz obrázků kdykoli. K zavedení obrazu obrázku ze souboru voláme metodu **LoadFromFile** obrázku a k uložení obrazu metodu **SaveToFile**. **LoadFromFile** a **SaveToFile** přebírají jméno souboru jako parametr. **LoadFromFile** používá příponu souboru k určení, který typ grafického objektu má být vytvořen a zaveden. **SaveToFile** ukládá typ souboru určený typem ukládaného grafického objektu. K zavedení bitové mapy do ovladače **Image**, např. předáme jméno souboru bitové mapy metodě **LoadFromFile** obrázku:

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  Image1->Picture->LoadFromFile("RANDOM.BMP");
}
```

Obrázek používající BMP jako standardní příponu pro soubory bitových map, vytváří svou grafiku jako **TBitmap** a pak volají metodu **LoadFromFile** obrázku. Jelikož grafika je bitová mapa, je zaveden její obraz ze souboru jako bitová mapa.

18. Když pracujeme na zařízení založené na paletě, pak Builder automaticky řídí podporu realizace palety. Tj. jestliže máme ovladač, který má paletu, můžeme používat dvě metody zděděné od **TControl** k řízení jak Windows přizpůsobí tuto paletu. Podpora palety pro ovladače má tyto dva aspekty: specifikace palety pro ovladač a reakce na změny palety. Mnoho ovladačů paletu nepoužívá. Nicméně ovladače, které obsahují grafiku (jako je např. komponenta **Image**) ji mít musí k interakci s Windows a ovladači řízení obrazovky k zajištění odpovídajícího vzhledu ovladače. Windows se odkazuje na tento proces jako na *realizaci palet*. Realizace palet je proces zajišťující, že nejvrchnější okno používá svou plnou paletu a spodní okna používají z této palety to co je možné a ostatní barvy mapují na barvy v „reálné“ paletě. Když se okno přesune nad jiné, Windows stále realizuje palety. Builder sám neposkytuje specifickou podporu pro vytváření nebo obhospodařování palet, jiných než bitových map. Jestliže máme paletu, kterou chceme použít na ovladač, musíme říci naši aplikaci aby použila tuto paletu. Pro specifikaci palety pro ovladač, předefinujeme metodu **GetPalette** ovladače, aby vracela madlo požadované palety. Specifikací palety pro ovladač provedeme v naši aplikaci dvě věci: řekneme aplikaci, že paleta našeho ovladače má být realizována a určíme paletu použitou pro realizaci.

    Jestliže náš ovladač specifikuje paletu předefinováním **GetPalette**, Builder automaticky přebírá odpovědnost za reakce na zprávy palety od Windows. Metoda provádějící správu palet je **PaletteChanged**. Pro normální operace, nepotřebujeme nikdy změnit její implicitní chování. Primární úloha **PaletteChanged** je v určení, zda realizovaná paleta ovladače je na popředí nebo na pozadí. Windows ovládá tuto realizaci palet tím, že nejvrchnější okno má paletu popředí a ostatní okna získají palety pozadí. Builder provádí jeden krok navíc, realizuje palety pro ovladače v Tab pořadí oken. Toto implicitní chování můžeme chtít předefinovat pouze, když chceme aby ovladač, který není první v Tab pořadí, získal paletu popředí.

19. Když kreslíme složitější obrázek obecnou technikou v programování Windows, vytvoříme neobrazovkovou bitovou mapu, nakreslíme na ní obrázek a potom překopírujeme kompletní obraz na definitivní místo na obrazovce. Použití obrázku neobrazovkové bitové mapy redukuje mihotání způsobené opakovaným kreslením přímo na obrazovku. Objekty bitových map v Builderu reprezentující bitově mapované obrázky ve zdrojích a souborech mohou také pracovat jako obrázky neobrazovkových bitových map. Když vytváříme složitější grafický obrázek, nemusíme obecně kreslit přímo na plátno, které je zobrazováno na obrazovce. Namísto kreslení na plátno formuláře nebo ovladače, můžeme vytvořit objekt bitové mapy, kreslit na jeho plátno a potom překopírovat kompletní obraz na plátno obrazovky.

    Na příklad kreslení složitějšího obrázku na neobrazovkovou bitovou mapu se můžeme podívat do zdrojového kódu ovladače **Gauge** ze stránky *Samples* Palety komponent. Tento ovladač kreslí různé tvary a texty na neobrazovkovou bitovou mapu před překopírováním na obrazovku. Delphi umožňuje čtyři různé způsoby kopírování obrázků z jednoho plátna na jiné. V závislosti na požadovaném efektu voláme různé metody:

Vytváří efekt	Voláme metodu
Kopírování celé grafiky	Draw
Kopírování a změna velikosti	StretchDraw
Kopírování části plátna	CopyRect
Kopírování bitových map s rastrovou operací	BrushCopy

    V nápovědě se můžeme podívat na příklady použití těchto metod.

20. Všechny grafické objekty, včetně pláten a jejich vlastních objektů (per, štětců a písem) mají události reakcí na změny v objektu. Použitím těchto událostí, můžeme umožnit naši komponentě (nebo aplikaci, která ji

používá) reagovat na změny překreslením svých obrazů. Pro reakci na změny v grafickém objektu, přiřadíme metodu události **OnChange** objektu. S tímto se seznámíme v následujícím zadání.

21. Jednoduchým typem komponenty je grafický ovladač. Protože grafický ovladač nikdy nemůže získat vstupní zaostření, nemá a nepotřebuje madlo okna. Uživatel aplikace obsahující grafické ovladače může stále manipulovat s ovladači pomocí myši, ale nemůže použít klávesnici. Grafická komponenta vytvářená v tomto bodě odpovídá komponentě **TShape** ze stránky *Additional* Palety komponent. Přestože vytvářená komponenta je identická, je nutno ji nazvat jinak, aby nedošlo k duplikaci identifikátorů. Naši komponentu nazveme **TSampleShape**. Vytvoření grafické komponenty vyžaduje tři kroky: vytvoření a registraci komponenty, zveřejnění zděděných vlastností a přidání grafických možností. Vytváření každé komponenty začíná vždy stejně. V našem příkladě použijeme manuální vytváření s těmito specifikami: jednotku komponenty nazveme *Shapes*, odvodíme nový typ komponenty **TSampleShape** od **TGraphicControl** a registrujeme **TSampleShape** na stránce *Samples* Palety komponent. Výsledkem této práce je:

```
#ifndef ShapesH
#define ShapesH
#include <vcl\SysUtils.hpp>
#include <vcl\Controls.hpp>
#include <vcl\Classes.hpp>
#include <vcl\Forms.hpp>
class TSampleShape : public TGraphicControl
{
private:
protected:
public:
__published:
};
#endif

#include <vcl\vcl.h>
#pragma hdrstop
#include "Shapes.h"
namespace Shapes
{
  void __fastcall Register()
  {
    TComponentClass classes[1] = {__classid(TSampleShape)};
    RegisterComponents("Samples", classes, 0);
  }
}
```

Když odvodíme typ komponenty, můžeme určit, které vlastnosti a události deklarované v chráněné části typu předka chceme zpřístupnit pro uživatele naši nové komponenty. Potomci **TGraphicControl** vždy zveřejňuje všechny vlastnosti, které umožňují komponentě aby pracovala jako ovladač, tj. musíme zveřejnit všechny reakce na události myši a obsluhu tažení. Zveřejňování zděděných vlastností a událostí spočívá v opětovné deklaraci jména vlastnosti ve zveřejňované části deklarace typu objektu. V našem příkladě zveřejníme tři události myši, tři události tažení a dvě vlastnosti tažení:

```
class TSampleShape : public TGraphicControl
{
__published:
  __property DragCursor;
  __property DragMode;
  __property OnDragDrop;
  __property OnDragOver;
  __property OnEndDrag;
  __property OnMouseDown;
  __property OnMouseMove;
  __property OnMouseUp;
};
```

Tím ovladač zpřístupní pro své uživatele práci s myší a tažení. Když máme deklarovanou svou grafickou komponentu a zveřejněné některé potřebné zděděné vlastnosti, můžeme naši komponentě přidat požadované grafické možnosti. Při vytváření grafického ovladače vždy provádíme tyto dva kroky: určíme co kreslit a nakreslíme obraz komponenty. Dále pro náš příklad musíme přidat některé vlastnosti, které umožní vývojáři aplikace použít náš ovladač k přizpůsobení vzhledu při návrhu. Grafický ovladač má obecně možnost měnit svůj vzhled v reakci na dynamické nebo uživatelem specifikované podmínky. Grafická komponenta, která je

stále stejná může být tvořena importovaným obrazem bitové mapy a nemusí to být grafický ovladač. Obecně, vzhled grafického ovladače závisí na některých kombinacích hodnot svých vlastností. Např. ovladač **Gauge** má vlastnosti které určují jeho tvar. Podobně ovladač **Shape** má vlastnost, která určuje jaký typ tvaru bude kreslen. Pro přidáni možnosti ovladači **Shape** určit kreslený tvar, přidáme vlastnost nazvanou **Shape**, což provedeme v těchto třech krocích: deklarujeme typ vlastnosti, deklarujeme vlastnost a zapíšeme implementaci metody. Když deklarujeme vlastnost uživatelem definovaného typu, musíme nejprve deklarovat typ vlastnosti a teprve potom objektový typ, který obsahuje vlastnost. Většina uživatelem definovaných typů pro vlastnosti jsou výčtové typy. Pro ovladač **Shape**, použijeme výčtový typ s prvky pro každý typ tvaru, která ovladač může kreslit. Přidáme následující definici typu před deklaraci třídy **Shape**:

```
enum TSampleShapeType { sstRectangle, sstSquare, sstRoundRect,
                        sstRoundSquare, sstEllipse, sstCircle };
class TSampleShape : public TGraphicControl
```

Nyní můžeme tento typ použít k deklarování nové vlastnosti v objektu. Když deklarujeme vlastnost, obvykle potřebujeme deklarovat soukromou položku k uložení dat vlastnosti, a specifikuje přístupové metody vlastnosti (čtení hodnoty provádíme často přímo). Pro náš ovladač deklarujeme položku pro uložení aktuálního tvaru a deklarujeme vlastnost, která čte tuto položku a zapisuje ji prostřednictvím volání metody. Přidáme tedy do deklarace typu **TSampleShape** toto:

```
class TSampleShape : public TGraphicControl
{
private:
  TSampleShapeType FShape;
  void __fastcall SetShape(TSampleShapeType Value);
__published:
  __property TSampleShapeType Shape={read=FShape,write=SetShape,nodefault};
};
```

Nyní ještě musíme implementovat metodu **SetShape**. Do souboru CPP jednotky přidáme:

```
void __fastcall TSampleShape::SetShape(TSampleShapeType Value)
{
  if (FShape != Value){
    FShape = Value;
    Invalidate();
  }
}
```

K umožnění změny implicitních vlastností a inicializaci vlastněných objektů pro naši komponentu musíme předefinovat zděděný konstruktor a destruktor. V obou nesmíme zapomenou volat zděděnou metodu. Implicitní velikost grafického ovladače je malá a tak změníme šířku a výšku v konstruktoru. V našem příkladě nastavíme velikost obou rozměrů na 65 bodů. Do deklarace třídy komponenty přidáme předefinování konstruktoru:

```
class TSampleShape : public TGraphicControl
{
public:
  virtual __fastcall TSampleShape(TComponent* Owner);
}
```

opětovně deklarujeme vlastnosti **Height** a **Width** s jejich novými implicitními hodnotami:

```
class TSampleShape : public TGraphicControl
{
__published:
  __property Height = {default=65};
  __property Width = {default=65};
};
```

a do souboru CPP zapíšeme implementaci nového konstruktoru:

```
__fastcall TSampleShape::TSampleShape(TComponent* Owner)
    : TGraphicControl(Owner)
{
  Width = 65;
  Height = 65;
}
```

Implicitně plátno má tenké černé pero a plný bílý štětec. Pro povolení změny těchto prvků plátna vývojáři používajícího ovladač **Shape**, musíme poskytnout objekty pro manipulaci s nimi při návrhu, a potom kopírovat tyto objekty na plátno, když kreslíme. Objekty jako je pero nebo štětec se nazývají vlastněné objekty, protože komponenta je vlastní a je zodpovědná za jejich vytvoření a zrušení. Spravování vlastněných objektů

vyžaduje tři kroky: deklaraci položek objektu, deklaraci přístupových vlastností a inicializaci vlastněných objektů. Každý objekt vlastněný komponentou musí mít objektovou položku deklarovanou v komponentě. Položka zajišťuje, že komponenta má vždy ukazatel na vlastněný objekt a může tak před svým zrušením objekt zrušit. Obecně komponenta inicializuje vlastněné objekty ve svém konstruktoru a ruší ve svém destruktoru. Položky pro vlastněné objekty jsou téměř vždy deklarovány jako soukromé. Jestliže uživatel komponenty požaduje přístup k vlastněným objektům, musí deklarovat vlastnosti, které poskytují přístup. Přidáme položky pro objekty pera a štětce k typu ovladače **Shape**:

```
class TSampleShape : public TGraphicControl
{
private:
  TPen *FPen;
  TBrush *FBrush;
};
```

K vlastněným objektům komponenty můžeme poskytnout přístup deklarací vlastností typu objektů. To dává vývojáři možnost přístupu k objektům jednak během návrhu, tak i při běhu aplikace. Obecně čtecí část vlastnosti je odkaz na položku objektu, ale zápisová část volá metodu, která povoluje komponentě reagovat na změny ve vlastněném objektu. V naši komponentě přidáme vlastnosti, které poskytují přístup k položkám pera a štětce. Musíme také deklarovat metody provádějící změny pera a štětce.

```
class TSampleShape : public TGraphicControl
{
private:
  TPen *FPen;
  TBrush *FBrush;
  void __fastcall SetBrush(TBrush *Value);
  void __fastcall SetPen(TPen *Value);
__published:
  __property TBrush* Brush={read=FBrush,write=SetBrush,nodefault};
  __property TPen* Pen={read=FPen,write=SetPen,nodefault};
};
```

Potom do souboru CPP jednotky zapíšeme metody **SetBrush** a **SetPen**:

```
void __fastcall TSampleShape::SetBrush(TBrush* Value)
{
  FBrush->Assign(Value);
}
void __fastcall TSampleShape::SetPen(TPen* Value)
{
  FPen->Assign(Value);
}
```

Jestliže k naši komponentě přidáme objekty, konstruktor komponenty musí tyto objekty inicializovat a tak umožnit uživateli pracovat s objekty při běhu aplikace. Podobně destruktor komponenty musí také uvolnit vlastněné objekty před uvolněním komponenty samotné. V konstruktoru ovladače **Shape** vytvoříme pero a štětec

```
__fastcall TSampleShape::TSampleShape(TComponent* Owner)
    : TGraphicControl(Owner)
{
  Width = 65;
  Height = 65;
  FBrush = new TBrush();
  FPen = new TPen();
}
```

přidáme předefinování destruktoru do deklarace objektu komponenty

```
class TSampleShape : public TGraphicControl
{
public:
  virtual __fastcall TSampleShape(TComponent* Owner);
  __fastcall ~TSampleShape();
};
```

a do souboru CPP zapíšeme nový destruktor:

```
__fastcall TSampleShape::~TSampleShape()
{
  delete FPen;
  delete FBrush;
```

```
}
```

Jako poslední krok ve zpracování objektů pera a štětce, musíme zajistit, že změny v peru a štětci způsobí překreslení samotného ovladače **Shape**. Objekty pera i štětce mají události **OnChange** a můžeme tedy v ovladači **Shape** vytvořit metodu a přiřadit ji oběma událostem **OnChance**. V našem kódu se změní:

```
class TSampleShape : public TGraphicControl
{
public:
  void __fastcall StyleChanged(TObject* Owner);

__fastcall TSampleShape::TSampleShape(TComponent* Owner)
    : TGraphicControl(Owner)
{
  Width = 65;
  Height = 65;
  FBrush = new TBrush();
  FBrush->OnChange = StyleChanged;
  FPen = new TPen();
  FPen->OnChange = StyleChanged;
}
void __fastcall TSampleShape::StyleChanged(TObject* Owner)
{
  Invalidate();
}
```

Po provedení těchto změn se komponenta v reakci na změnu pera nebo štětce překreslí.

Základní možností grafického ovladače je schopnost nakreslení svého obrazu na obrazovku. Abstraktní typ **TGraphicControl** definuje virtuální metodu nazvanou **Paint**, kterou předefinujeme k nakreslení požadovaného obrazu naší komponenty. Metoda **Paint** pro komponentu **Shape** musí provést několik věcí: použít pero a štětec vybraný uživatelem, použit vybraný tvar a upravit souřadnice tak, aby čtverec a kruh měli stejnou šířku a výšku. Předefinování metody **Paint** vyžaduje přidat **Paint** do deklarace komponenty a do souboru CPP zapsat metodu **Paint**. Po provedení těchto věcí dostaneme:

```
class TSampleShape : public TGraphicControl
{
protected:
  virtual void __fastcall Paint();
};

void __fastcall TSampleShape::Paint()
{
  int X, Y, W, H, S;
  Canvas->Pen = FPen;
  Canvas->Brush = FBrush;
  W = Width;
  H = Height;
  X = Y = 0;
  if (W < H) S = W;
  else S = H;
  switch (FShape) {
    case sstSquare:
    case sstRoundSquare:
    case sstCircle:
      X = (W - S)/2;
      Y = (H - S)/2;
      break;
    default:
      break;
  }
  switch (FShape) {
    case sstSquare:
      W = H = S;
    case sstRectangle:
      Canvas->Rectangle(X,Y,X+W,Y+H);
      break;
    case sstRoundSquare:
```

```
      W = H = S;
    case sstRoundRect:
      Canvas->RoundRect(X,Y,X+W,Y+H,S/4,S/4);
      break;
    case sstCircle:
      W = H = S;
    case sstEllipse:
      Canvas->Ellipse(X,Y,X+W,Y+H);
      break;
    default:
      break;
  }
}
```

Tím je vývoj této komponenty dokončen.

22. Dále se pokusíme vytvořit komponentu digitálních hodin. Protože budeme provádět určitý textový výstup, můžeme odvození provést od komponenty **Label**. V tomto případě může ale uživatel měnit titulek komponenty. Abychom tomu zabránili, použijeme jako rodičovskou třídu komponentu **TCustomLabel**, která má stejné schopnosti, ale méně zveřejňovaných vlastností (můžeme sami určit, které vlastnosti zveřejníme). Kromě předeklarování některých vlastností třídy předka bude mít naše komponenta (**TDigClock**) jednu vlastní a to vlastnost **Active**. Tato vlastnost udává, zda hodiny pracují či ne. Komponenta hodin bude uvnitř obsahovat komponentu **Timer**, která ji nutí pracovat. Časovač ale není veřejný přes vlastnost, protože nechceme aby byl přístupný. Pouze jeho vlastnost **Enabled** je zaobalena do naši vlastnosti **Active**. Následuje výpis programové jednotky nové komponenty (nejprve je uveden výpis hlavičkového souboru):

```
#ifndef DigClockH
#define DigClockH
#include <vcl\SysUtils.hpp>
#include <vcl\Controls.hpp>
#include <vcl\Classes.hpp>
#include <vcl\Forms.hpp>
#include <vcl\StdCtrls.hpp>
class TDigClock : public TCustomLabel
{
private:
  TTimer *FTimer;
  bool __fastcall GetActive();
  void __fastcall SetActive(bool Value);
protected:
  void __fastcall UpdateClock(TObject *Sender);
public:
  __fastcall TDigClock(TComponent* Owner);
  __fastcall ~TDigClock();
__published:
  __property Align;
  __property Alignment;
  __property Color;
  __property Font;
  __property ParentColor;
  __property ParentFont;
  __property ParentShowHint;
  __property PopupMenu;
  __property ShowHint;
  __property Transparent;
  __property Visible;
  __property bool Active = {read=GetActive, write=SetActive, nodefault};
};
#endif

#include <vcl\vcl.h>
#pragma hdrstop
#include "DigClock.h"
__fastcall TDigClock::TDigClock(TComponent* Owner) : TCustomLabel(Owner)
{
  TComponentClass classes[1] = {__classid(TTimer)};
```

```
  RegisterClasses(classes, 0);
  FTimer = new TTimer(Owner);
  FTimer->OnTimer = UpdateClock;
  FTimer->Enabled = true;
}
__fastcall TDigClock::~TDigClock()
{
  delete FTimer;
}
void __fastcall TDigClock::UpdateClock(TObject *Sender)
{
  Caption = TimeToStr(Time());
}
bool __fastcall TDigClock::GetActive()
{
  return FTimer->Enabled;
}
void __fastcall TDigClock::SetActive(bool Value)
{
  FTimer->Enabled = Value;
}
namespace Digclock
{
  void __fastcall Register()
  {
    TComponentClass classes[1] = {__classid(TDigClock)};
    RegisterComponents("Samples", classes, 0);
  }
}
```

Prostudujte si uvedený výpis. Před vytvořením objektu **Timer** je požadována registrace typu této komponenty, která je naší komponentou používána. Jinak by vám měl výpis být srozumitelný. Vyzkoušejte použití této komponenty v nějaké aplikaci.

23. Vytvořte komponentu analogových hodin.
24. Zápis komponenty a jejich vlastností, metod a událostí je pouze částí procesu vytváření komponenty. Musíme ještě umožnit, abychom s komponentou mohli manipulovat při návrhu. To vyžaduje provedení těchto kroků: registrování komponenty v Builderu, přidání bitové mapy komponenty na paletu, poskytnutí nápovědy pro vlastnosti a události a uložení a zavádění vlastnosti. Ne všechny tyto kroky jsou potřebné pro každou komponentu. Např. jestliže nedefinujeme nové vlastnosti nebo události nemusíme k nim vytvářet nápovědu. Nezbytná je vždy pouze registrace. Builder vyžaduje pro seskupování a umisťování komponent na Paletu komponent registraci komponenty. Registrace pracuje na základě programové jednotky, a jestliže vytvoříme několik komponent v jedné jednotce, registrujeme je najednou. S registrací komponent jsme se již seznámili.
25. Každá komponenta vyžaduje bitovou mapu reprezentující komponentu na Paletě komponent. Jestliže nespecifikujeme svou vlastní bitovou mapu, Builder použije implicitní. Jelikož bitové mapy palety jsou potřebné pouze během návrhu, nejsou přeloženy v jednotce komponenty. Jsou v souboru zdrojů Windows se stejným jménem jako má jednotka, ale s příponou DCR (Dynamic Component Resource). Tento soubor zdrojů můžeme vytvořit pomocí Editoru obrázků v Builderu. Každá bitová mapa je čtverec o straně 24 bodů. Pro každou jednotku, kterou chceme instalovat, potřebujeme soubor DCR a v každém souboru DCR potřebujeme bitovou mapu pro každou registrovanou komponentu. Obraz bitové mapy má stejné jméno jako komponenta. Soubor DCR musí být ve stejném adresáři jako jednotka komponenty, Builder zde tento soubor hledá, když instaluje komponenty na paletu. Např. jestliže vytvoříme komponentu nazvanou **TMujOvladac** v jednotce nazvané *ToolBox*, musíme vytvořit soubor zdrojů nazvaný TOOLBOX.DCR, který obsahuje bitovou mapu nazvanou TMUJOVLADAC. Ve jménech zdrojů nezáleží na velikostí písmen, ale podle konvence je obvykle zapisujeme velkými písmeny. Pokuste se vytvořit bitovou mapu pro některou komponentu, kterou jste již udělali.
26. Když vybereme komponentu na formuláři, případně vlastnost nebo událost v Inspektoru objektů, pak můžeme stisknutím F1 získat informaci o tomto prvku. Uživatelé našich komponent mohou získat stejné informace o naší komponentě, jestliže vytvoříme příslušné soubory nápovědy. Můžeme poskytnout malý soubor nápovědy s informacemi o našich komponentách a uživatelé mohou nalézt naši dokumentaci bez nutnosti nějakého speciálního kroku. Naše nápověda se stane částí nápovědného systému Builderu. K vytvoření souboru nápovědy můžeme použít libovolný nástroj. Builder obsahuje Microsoft Help Workshop, který mů-

žeme použít k vytvoření našeho nápovědného souboru. Při vytváření nápovědného souboru je nutno dodržovat tyto konvence: Každá komponenta musí mít obrazovku (obsahuje stručný popis a seznam všech vlastností, událostí a metod dostupných uživateli; obrazovka komponenty musí být označena poznámkou pod čarou K, která obsahuje jméno komponenty, např. *TMemo*). Každá vlastnost, událost a metoda deklarovaná v komponentě musí mít obrazovku. Každá obrazovka komponenty, vlastnosti, události nebo metody musí mít unikátní ID, které je zadáno jako poznámka pod čarou #, název zadaný jako poznámka pod čarou $ a klíčové slovo používané při hledání jako poznámku pod čarou K.

27. Builder ukládá formuláře a jejich komponenty v souborech formulářů (DFM). Soubor formuláře je binární reprezentace vlastností formuláře a jeho komponent. Když uživatel Builderu přidává komponenty, zapíše je na jejich formulář, a naše komponenty musí mít schopnost zapsat své vlastnosti do souboru formuláře při uložení. Podobně, když provádíme aplikaci, komponenty se musí sami obnovit ze souboru formuláře. Když vývojář aplikace navrhuje formulář, Builder ukládá popis formuláře do souboru formuláře, který je později připojen k přeložené aplikaci. Když uživatel spustí aplikaci, je přečten tento popis. Popis formuláře obsahuje seznam vlastností formuláře společně s podrobným popisem všech komponent umístěných na formuláři. Každá komponenta, včetně samotného formuláře, je odpovědná za uložení a zavádění svého vlastního popisu. Implicitně při samotném ukládání, komponenta zapíše hodnoty všech svých veřejných a zveřejňovaných vlastností, které se liší od svých implicitních hodnot a to v pořadí jejich deklarací. Při zavádění, se komponenta nejdříve vytvoří sama, pak nastaví všechny vlastnosti na jejich implicitní hodnoty a nakonec čte uložené neimplicitní hodnoty vlastností. Tento implicitní mechanismus slouží mnoha komponentám a nevyžaduje žádnou akci od tvůrce komponent. Nicméně je několik způsobů, jak můžeme přizpůsobit proces ukládání a zavádění potřebný pro naši jistou komponentu.

    Komponenty Builderu ukládají hodnoty svých vlastností pouze, jestliže se tyto hodnoty liší od implicitních hodnot. Jestliže nespecifikujeme jinak, Builder předpokládá, že vlastnosti nemají implicitní hodnotu a jejích hodnota je tedy ukládána stále. Vlastnosti jejichž hodnota není nastavena v konstruktoru komponenty mají nulové hodnoty. Nulová hodnota znamená, že paměť vyhrazená pro vlastnost je vynulována. Tj. číselné hodnoty jsou nulové, logické hodnoty jsou nastaveny na *false*, ukazatele na NULL apod. Ke specifikaci implicitní hodnoty pro vlastnost, přidáme direktivu **default** a novou implicitní hodnotu na konec deklarace vlastnosti. Můžeme také specifikovat implicitní hodnotu při opětovné deklaraci vlastnosti. Jedním smyslem opětovné deklarace vlastnosti je změna implicitní hodnoty. Specifikace implicitní hodnoty nepřiřazuje automaticky tuto hodnotu vlastnosti při vytváření objektu. Musíme ji ještě přiřadit v konstruktoru objektu. S tím jsme se již seznámili. Můžeme také řídit, zda Builder ukládá každou vlastnost komponenty. Implicitně všechny vlastnosti ve zveřejňované části deklarace objektu jsou ukládány. Můžeme zvolit jejich neukládání nebo navrhnout funkci, která při běhu programu určí zda vlastnost ukládat. K řízení ukládání vlastnosti, přidáme direktivu **stored** k deklaraci vlastnosti, následovanou *true*, *false* nebo jménem metody typu **bool**. Direktivu **stored** můžeme použít i v opětovné deklaraci vlastnosti. Následující kód ukazuje komponentu, která deklaruje tři nové vlastnosti. První je vždy ukládána, druhá není ukládána nikdy a třetí je ukládána v závislosti na hodnotě metody

```
class TPrikladKomponenty : public TComponent
{
protected:
  bool __fastcall UkladatJi();
public:
  __property int Ukladat = { stored = true };
published:
  __propetty int Neukladat = { stored = false };
  __property int NekdyUkladat = { stored UkladatJi };
};
```

    Po přečtení všech hodnot vlastností komponenty z uloženého popisu je volána virtuální metoda nazvaná **Loaded**, která provádí požadované změny v inicializaci. Volání **Loaded** proběhne před zobrazením formuláře a jeho ovladačů. K inicializaci komponenty po zavedení hodnot jejich vlastností předefinujeme metodu **Loaded**. První věc, kterou v předefinované metodě **Loaded** musíme provést je volání zděděné metody **Loaded**. To zajistí, že všechny zděděné vlastnosti jsou správně inicializovány před provedením inicializace své vlastní komponenty.

28. Začneme s vývojem další komponenty. Builder poskytuje několik typů abstraktních komponent, které můžeme použít jako základ pro přizpůsobování komponent. V tomto bodě si ukážeme jak vytvořit malý jednoměsíční kalendář na základě komponenty mřížky **TCustomGrid**. Vytvoření kalendáře provedeme v sedmi krocích: vytvoření a registrace komponenty, zveřejnění zděděných vlastností, změnu inicializačních hodnot, změna velikosti buněk, vyplnění buněk, navigaci měsíců a roků a navigaci dní. Použijeme ruční postup vytváření a registrace komponenty s těmito specifikami: jednotku komponenty nazveme *CalSamp*, odvodíme

nový typ komponenty nazvaný **TSampleCalendar** od **TCustomGrid** a registrujeme **TSampleCalendar** na stránce *Samples* Palety komponent. Výsledkem této práce je (musíme přidat i hlavičkový soubor *Grids.hpp*):

```
#ifndef CALSAMPH
#define CALSAMPH
#include <vcl\SysUtils.hpp>
#include <vcl\Controls.hpp>
#include <vcl\Classes.hpp>
#include <vcl\Forms.hpp>
#include <vcl\Grids.hpp>
class TSampleCalendar : public TCustomGrid
{
private:
protected:
public:
__published:
};
#endif

#include <vcl\vcl.h>
#pragma hdrstop
#include "CALSAMP.h"
namespace Calsamp
{
  void __fastcall Register()
  {
    TComponentClass classes[1] = {__classid(TSampleCalendar)};
    RegisterComponents("Samples", classes, 0);
  }
}
```

Abstraktní komponenta mřížky **TCustomGrid** poskytuje velký počet chráněných vlastností. Můžeme zvolit, které z těchto vlastností chceme zpřístupnit v naši vytvářené komponentě. K zveřejnění zděděných chráněných vlastností, opětovně deklarujeme vlastnosti ve zveřejňované části deklarace naši komponenty. Pro kalendář zveřejníme následující vlastnosti a události:

```
class TSampleCalendar : public TCustomGrid
{
__published:
  __property Align;
  __property BorderStyle;
  __property Color;
  __property Ctl3D;
  __property Font;
  __property GridLineWidth;
  __property ParentColor;
  __property ParentCtl3D;
  __property ParentFont;
  __property OnClick;
  __property OnDblClick;
  __property OnDragDrop;
  __property OnDragOver;
  __property OnEndDrag;
  __property OnKeyDown;
  __property OnKeyPress;
  __property OnKeyUp;
};
```

Existuje ještě několik dalších vlastností, které jsou také zveřejňované, ale které pro kalendář nepotřebujeme. Příkladem je vlastnost **Options**, která umožňuje uživatelovi např. volit typ mřížky. Kalendář je mřížka s pevným počtem řádků a sloupců, i když ne všechny řádky vždy obsahují data. Z tohoto důvodu, jsme nezveřejnili vlastnosti mřížky **ColCount** a **RowCount**, neboť je jasné, že uživatel kalendáře nebude chtít zobrazovat nic jiného než sedm dní v týdnu. Nicméně musíme nastavit počáteční hodnoty těchto vlastností tak, aby týden měl vždy sedm dní. Ke změně počátečních hodnot vlastností komponenty, předefinujeme konstruktor a nastavíme požadované hodnoty. Musíme také předefinovat čirou metodu **DrawCell**. Dostaneme toto:

```
class TSampleCalendar : public TCustomGrid
```

```
{
protected:
  virtual void __fastcall DrawCell(long ACol, long ARow,
                      const Windows::TRect &Rect, TGridDrawState AState);
public:
  __fastcall TSampleCalendar(TComponent* Owner);
};

__fastcall TSampleCalendar::TSampleCalendar(TComponent* Owner)
    : TCustomGrid(Owner)
{
  ColCount = 7;
  RowCount = 7;
  FixedCols = 0;
  FixedRows = 1;
  ScrollBars = ssNone;
  Options = Options >> goRangeSelect << goDrawFocusSelected;
}
void __fastcall TSampleCalendar::DrawCell(long ACol, long ARow,
                      const Windows::TRect &Rect, TGridDrawState AState)
{
}
```

Nyní, když přidáme kalendář na formulář má sedm řádků a sedm sloupců s pevným horním řádkem. Pravděpodobně budeme chtít změnit velikost ovladače a udělat všechny buňky viditelné. Dále si ukážeme jak reagovat na zprávu změny velikosti od Windows k určení velikosti buněk.

Když uživatel nebo aplikace změní velikost okna nebo ovladače, Windows zasílá zprávu nazvanou WM_SIZE změněnému oknu nebo ovladači, které tak může nastavit svůj obraz na novou velikost. Naše komponenta může reagovat na tuto zprávu změnou velikosti buněk a zaplnit tak celou plochu ovladače. K reakci na zprávu WM_SIZE, přidáme do komponenty metodu reagující na zprávu. V našem případě ovladač kalendáře vyžaduje k reakci na WM_SIZE přidat chráněnou metodu nazvanou **WMSize** řízenou indexem zprávy WM_SIZE, a zapsat metodu, která vypočítá potřebné rozměry buněk, což umožní aby všechny buňky byly viditelné (více informací o zpracování zpráv Windows je uvedeno v bodech 30 až 33):

```
class TSampleCalendar : public TCustomGrid
{
protected:
  void __fastcall WMSize(TWMSize& Message);
__published:
  __property Align;
...
__property OnKeyUp;
  BEGIN_MESSAGE_MAP
    MESSAGE_HANDLER(WM_SIZE, TWMSize, WMSize);
  END_MESSAGE_MAP(TCustomGrid);
};

void __fastcall TSampleCalendar::WMSize(TWMSize &Message)
{
  int GridLines;
  GridLines = 6 * GridLineWidth;
  DefaultColWidth   = (Message.Width - GridLines) / 7;
  DefaultRowHeight  = (Message.Height - GridLines) / 7;
}
```

Nyní, jestliže přidáme kalendář na formulář a změníme jeho velikost, je vždy zobrazen tak, aby jeho buňky úplně zaplnili plochu ovladače. Zatím ale kalendář neobsahuje data.

Ovladač mřížky je zaplňován buňku po buňce. V případě kalendáře to znamená vypočítat datum (je-li) pro každou buňku. Implicitní zaplňování buněk provádíme virtuální metodou **DrawCell**. Knihovna běhu programu obsahuje pole s krátkými jmény dní a my je použijeme v nadpisu každého sloupce:

```
void __fastcall TSampleCalendar::DrawCell(long ACol, long ARow,
                             const TRect &ARect, TGridDrawState AState)
{
  String TheText;
  int TempDay;
  if (ARow == 0) TheText = ShortDayNames[ACol];
```

```
  Canvas->TextRect(ARect, ARect.Left + (ARect.Right - ARect.Left -
    Canvas->TextWidth(TheText)) / 2, ARect.Top + (ARect.Bottom -
    ARect.Top - Canvas->TextHeight(TheText)) / 2, TheText);
};
```

Pro ovladač kalendáře je vhodné, aby uživatel a aplikace měli mechanismus pro nastavování dne, měsíce a roku. Builder ukládá datum a čas v proměnné typu **TDateTime**. **TDateTime** je zakódovaná číselná reprezentace datumu a času, která je vhodná pro počítačové zpracování, ale není použitelná pro použití člověkem. Můžeme tedy ukládat datum v zakódovaném tvaru, poskytnout přístup k této hodnotě při běhu aplikace, ale také poskytnout vlastnosti **Day**, **Month** a **Year**, které uživatel komponenty může nastavit při návrhu. K uložení data pro kalendář, potřebujeme soukromou položku k uložení data a vlastnosti běhu programu, které poskytují přístup k tomuto datu. Přidání interního data ke kalendáři vyžaduje tři kroky: V prvním deklarujeme soukromou položku k uložení data:

```
class TSampleCalendar : public TCustomGrid
{
private:
  TDateTime FDate;
};
```

V druhém inicializujeme datovou položku v konstruktoru:

```
__fastcall TSampleCalendar::TSampleCalendar(TComponent* Owner)
    : TCustomGrid(Owner)
{
  ...
  FDate = FDate.CurrentDate();
}
```

V posledním deklarujeme vlastnost běhu programu k poskytnutí přístupu k zakódovaným datům. Potřebujeme metodu pro nastavení data, protože nastavení data vyžaduje aktualizaci obrazu ovladače na obrazovce:

```
class TSampleCalendar : public TCustomGrid
{
private:
  void __fastcall SetCalendarDate(TDateTime Value);
public:
  __property TDateTime CalendarDate={read=FDate,write=SetCalendarDate,nodefault};
};

void __fastcall TSampleCalendar::SetCalendarDate(TDateTime Value)
{
  FDate = Value;
  Refresh();
}
```

Zakódované datum je vhodné pro aplikaci, ale lidé dávají přednost práci s dny, měsíci a roky. Můžeme poskytnout alternativní přístup k těmto prvkům uložených zakódovaných dat vytvořením vlastností. Protože každý prvek dat (den, měsíc a rok) je celé číslo a jelikož nastavení každého z nich vyžaduje dekódování dat, můžeme se vyhnout opakování kódu sdílením implementačních metod pro všechny tři vlastnosti. Tj. můžeme zapsat dvě metody, první pro čtení prvku a druhou pro jeho zápis, a použít tyto metody k získání a nastavování všech tří vlastností. Deklarujeme tři vlastnosti a každé přiřadíme jedinečný index:

```
class TSampleCalendar : public TCustomGrid
{
public:
  __property int Day = {read=GetDateElement, write=SetDateElement,
                        index=3, nodefault};
  __property int Month = {read=GetDateElement, write=SetDateElement,
                          index=2, nodefault};
  __property int Year = {read=GetDateElement, write=SetDateElement,
                         index=1, nodefault};
```

Dále zapíšeme deklarace a definice přístupových metod, pracujících s hodnotami podle hodnoty indexu:

```
class TSampleCalendar : public TCustomGrid
{
private:
  int __fastcall GetDateElement(int Index);
  void __fastcall SetDateElement(int Index, int Value);
}
```

```
int __fastcall TSampleCalendar::GetDateElement(int Index)
{
  unsigned short AYear, AMonth, ADay;
  int result;
  FDate.DecodeDate(&AYear, &AMonth, &ADay);
  switch (Index) {
    case 1: result = AYear; break;
    case 2: result = AMonth; break;
    case 3: result = ADay; break;
    default: result = -1;
  }
  return result;
}
void __fastcall TSampleCalendar::SetDateElement(int Index, int Value)
{
  unsigned short AYear, AMonth, ADay;
  if (Value > 0) {
    FDate.DecodeDate(&AYear, &AMonth, &ADay);
    switch (Index) {
      case 1: AYear = Value; break;
      case 2: AMonth = Value; break;
      case 3: ADay = Value; break;
      default: return;
    }
  }
  FDate = TDateTime(AYear, AMonth, ADay);
  Refresh();
}
```

Nyní můžeme nastavovat den, měsíc a rok kalendáře během návrhu použitím Inspektora objektů a při běhu aplikace použitím kódu. Zatím ještě ale nemáme přidaný kód pro zápis datumu do buněk. Přidání čísel do kalendáře vyžaduje několik úvah. Počet dní v měsíci závisí na tom, o který měsíc se jedná a zda daný rok je přestupný. Dále měsíce začínají v různém dni v týdnu, v závislosti na měsíci a roku. V předchozí části je popsáno jak získat aktuální měsíc a rok. Nyní můžeme určit, zda specifikovaný rok je přestupný a počet dní v měsíci. Objektu kalendáře přidáme dvě metody: logickou funkci indikující zda současný rok je přestupný a celočíselnou funkci vracející počet dní v současném měsíci. Obě deklarace metod musíme také přidat do deklarace typu **TSampleCalendar**.

```
bool __fastcall TSampleCalendar::IsLeapYear()
{
  return (Year % 4 == 0) && ((Year % 100 != 0) || (Year % 400 == 0));
}
int __fastcall TSampleCalendar::DaysThisMonth()
{
  int DaysPerMonth[12] = {31,28,31,30,31,30,31,31,30,31,30,31};
  int result;
  if ((int)FDate == 0) result = 0;
  else {
    result = DaysPerMonth[Month-1];
    if ((Month == 2) && (IsLeapYear())) result++;
  }
  return result;
}
```

Funkce **DaysThisMonth** vrací nulu, jestliže uložené datum je prázdné. To slouží aplikaci k indikování chybného data. Indikace, že měsíc nemá dny, vytvoří prázdný kalendář reprezentující chybné datum. Když již máme informace o přestupných rocích a dnech v měsíci, můžeme vypočítat, kde v mřížce je konkrétní datum. Výpočet je založen na dni v týdnu, kdy měsíc začíná. Protože potřebujeme ofset měsíce pro každou buňku, je praktičtější je vypočítat pouze, když měníme měsíc nebo rok. Tuto hodnotu můžeme uložit v položce třídy a aktualizovat ji při změně data. Zaplnění dnů do příslušných buněk provedeme takto: Přidáme položku ofsetu měsíce a metodu aktualizující hodnotu položky k objektu:

```
class TSampleCalendar : public TCustomGrid
{
private:
  int FMonthOffset;
protected:
```

```
  virtual void __fastcall UpdateCalendar();
};

void __fastcall TSampleCalendar::UpdateCalendar()
{
  unsigned short AYear, AMonth, ADay;
  TDateTime FirstDate;
  if ((int)FDate != 0) {
    FDate.DecodeDate(&AYear, &AMonth, &ADay);
    FirstDate = TDateTime(AYear, AMonth, 1);
    FMonthOffset = 1- FirstDate.DayOfWeek();
  }
  Refresh();
}
```

Přidáme příkazy do konstruktoru a metod **SetCalendarDate** a **SetDateElement**, které volají novou aktualizační metodu při změně data.

```
__fastcall TSampleCalendar::TSampleCalendar(TComponent* Owner)
    : TCustomGrid(Owner)
{
  ...
  UpdateCalendar();
}
void __fastcall TSampleCalendar::SetCalendarDate(TDateTime Value)
{
  FDate = Value;
  UpdateCalendar();
}
void __fastcall TSampleCalendar::SetDateElement(int Index, int Value)
{
  ...
  FDate = TDateTime(AYear, AMonth, ADay);
  UpdateCalendar();
}
```

Přidáme ke kalendáři metodu, která vrací číslo dne, když předáme souřadnice řádku a sloupce buňky:

```
int __fastcall TSampleCalendar::DayNum(int ACol, int ARow)
{
  int result = FMonthOffset + ACol + (ARow - 1) * 7;
  if ((result < 1) || (result > DaysThisMonth())) result = -1;
  return result;
}
```

Nesmíme zapomenou přidat deklaraci **DayNum** do deklarace třídy komponenty. Nyní, když již víme v které buňce které datum je, můžeme doplnit **DrawCell** k plnění buňky datem:

```
void __fastcall TSampleCalendar::DrawCell(long ACol, long ARow,
                          const TRect &ARect, TGridDrawState AState)
{
  String  TheText;
  int TempDay;
  if (ARow == 0) TheText = ShortDayNames[ACol];
  else {
    TheText = "";
    TempDay = DayNum(ACol, ARow);
    if (TempDay != -1) TheText = IntToStr(TempDay);
  }
  Canvas->TextRect(ARect, ARect.Left + (ARect.Right - ARect.Left -
    Canvas->TextWidth(TheText)) / 2, ARect.Top + (ARect.Bottom -
    ARect.Top - Canvas->TextHeight(TheText)) / 2, TheText);
};
```

Jestliže nyní opětovně instalujeme komponentu a umístíme ji na formulář, vidíme správné informace pro současný měsíc.

Nyní, když již máme čísla v buňkách kalendáře, je vhodné přesunout vybranou buňku na buňku se současným datem. Implicitně výběr začíná v levé horní buňce, a tak je potřeba nastavit vlastnosti **Row** a **Col**, když vytváříme kalendář a když změníme datum. K nastavení výběru na tento den, změníme metodu **UpdateCalendar** tak, aby nastavila obě vlastnosti před voláním **Refresh**:

```
void __fastcall TSampleCalendar::UpdateCalendar()
{
  unsigned short AYear, AMonth, ADay;
  TDateTime FirstDate;
  if ((int)FDate != 0) {
    FDate.DecodeDate(&AYear, &AMonth, &ADay);
    FirstDate = TDateTime(AYear, AMonth, 1);
    FMonthOffset = 1- FirstDate.DayOfWeek();
    Row = (ADay - FMonthOffset) / 7 + 1;
    Col = (ADay - FMonthOffset) % 7;
  }
  Refresh();
}
```

Vlastnosti jsou užitečné pro manipulace s komponentami, obzvláště během návrhu. Jsou ale typy manipulací, které často ovlivňují více než jednu vlastnost, a je tedy užitečné pro ně vytvořit metodu. Příkladem takového manipulace je služba kalendáře „následující měsíc“. Zpracování měsíce v rámci měsíců a případná inkrementace roku je jednoduchá, ale velmi výhodná pro vývojáře používající komponentu. Jedinou nevýhodou zaobalení manipulací do metody je, že metody jsou přístupné pouze za běhu aplikace. Pro kalendář přidáme následující čtyři metody pro následující a předchozí měsíc a rok:

```
void __fastcall TSampleCalendar::NextMonth()
{
  if (Month < 12) Month++;
  else {Year++; Month = 1;}
}
void __fastcall TSampleCalendar::PrevMonth()
{
  if (Month > 1) Month--;
  else {Year--; Month = 12;}
}
void __fastcall TSampleCalendar::NextYear()
{
  Year++;
}
void __fastcall TSampleCalendar::PrevYear()
{
  Year--;
}
```

Musíme také přidat deklarace nových metod k deklaraci třídy kalendáře. Nyní, když vytváříme aplikaci, která používá komponentu kalendáře, můžeme snadno implementovat procházení přes měsíce nebo roky.

K daném měsíci jsou možné dva způsoby navigace přes dny. První je použití kurzorových kláves a druhý je reakce na kliknutí myši. Standardní komponenta mřížky zpracovává oboje jako kliknutí. Tj. použití kurzorové klávesy je chápáno jako kliknutí na odpovídající buňku. Zděděné chování mřížky zpracovává přesun výběru v reakci na stisknutí kurzorové klávesy nebo kliknutí, ale jestliže chceme změnit vybraný den, musíme toto implicitní chování modifikovat. K obsluze přesunu v kalendáři, předefinujeme metodu **Click** mřížky. Když předefinováváme metodu jako je **Click**, musíme vždy vložit volání zděděné metody, a neztratit tak standardní chování. Následuje předefinovaná metoda **Click** pro mřížku kalendáře. Nesmíme zapomenout přidat deklaraci **Click** do **TSampleCalendar**:

```
void __fastcall TSampleCalendar::Click()
{
  TCustomGrid::Click();
  int TempDay = DayNum(Col, Row);
  if (TempDay != -1) Day = TempDay;
}
```

Nyní, když uživatel může změnit datum v kalendáři, musíme zajistit, aby aplikace mohla reagovat na tuto změnu. Do **TSampleCalendar** přidáme událost **OnChange**. Musíme deklarovat událost, položku k uložení události a virtuální metodu k volání události:

```
class TSampleCalendar : public TCustomGrid
{
private:
  TNotifyEvent FOnChange;
protected:
  virtual void __fastcall Change();
```

```
__published:
  __property TNotifyEvent OnChange = {read=FOnChange, write=FOnChange};
};
```

Dále zapíšeme metodu **Change**:

```
void __fastcall TSampleCalendar::Change()
{
  if (FOnChange != NULL) FOnChange(this);
}
```

Na konec metod **SetCalendarDate** a **SetDateElement** musíme ještě přidat příkaz volání metody **Change**:

```
void __fastcall TSampleCalendar::SetCalendarDate(TDateTime Value)
{
  FDate = Value;
  UpdateCalendar();
  Change();
}
void __fastcall TSampleCalendar::SetDateElement(int Index, int Value)
{
  ...
  FDate = TDateTime(AYear, AMonth, ADay);
  UpdateCalendar();
  Change();
}
```

Aplikace používající komponentu může nyní reagovat na změny data komponenty připojením obsluhy k události **OnChange**.

Když přecházíme po dnech v kalendáři, zjistíme nesprávné chování při výběru prázdné buňky. Kalendář umožňuje přesunutí na prázdnou buňku, ale nemění datum v kalendáři. Nyní zakážeme výběr prázdných buněk. K určení, zda daná buňka je vybíratelná, předefinujeme metodu **SelectCell** mřížky. **SelectCell** je funkce, která jako parametry přebírá číslo řádku a sloupce a vrací logickou hodnotu indikující zda specifikovaná buňka je vybíratelná. Metoda **SelectCell** bude nyní vypadat takto:

```
bool __fastcall TSampleCalendar::SelectCell(long ACol, long ARow)
{
  if (DayNum(ACol, ARow) == -1) return false;
  else return TCustomGrid::SelectCell(ACol, ARow);
}
```

Tím jsme dokončili tvorbu naší komponenty. Pokud první den v měsíci je neděle, pak první řádek naší komponenty je prázdný. Pokuste se odstranit tuto chybu.

29. Když pracujeme s připojenou databází, je často užitečné mít ovladač, závislý na této databázi. Aplikace tedy může založit propojení mezi ovladačem a nějakou částí databáze. Builder obsahuje závislá editační okna, seznamy, kombinovaná okna a mřížky. Můžeme také vytvořit svůj vlastní závislý ovladač. Je několik stupňů závislosti dat. Nejjednodušší je datová závislost pouze pro čtení, nebo prohlížení dat, a umožnit reakci na aktuální stav databáze. Složitější je editovatelná datová závislost, kde uživatel může editovat hodnoty v databázi manipulací s ovladačem. Nyní se pokusíme vytvořit ovladač určený pouze pro čtení, který je spojen s jednou položkou v databázi. Ovladač bude používat kalendář vytvořený v předchozím bodě. Vytvoření ovladače datově závislého kalendáře provedeme v následujících krocích: vytvoříme a registrujeme komponentu, uděláme ovladač určeny pouze pro čtení, přidáme datové propojení a reakce na změny dat.

    Použijeme obecný postup s těmito specifikami: jednotku komponenty nazveme *DBCal*, odvodíme nový typ komponenty nazvaný **TDBCalendar** od **TSampleCalendar** a registrujeme **TDBCalendar** na stránce *Samples* Palety komponent. Výsledek naší práce je:

```
#ifndef DBCalH
#define DBCalH
#include <vcl\SysUtils.hpp>
#include <vcl\Controls.hpp>
#include <vcl\Classes.hpp>
#include <vcl\Forms.hpp>
#include <vcl\Grids.hpp>
#include "CALSAMP.h"
class TDBCalendar : public TSampleCalendar
{
private:
protected:
public:
```

```
__published:
};
#endif

#include <vcl\vcl.h>
#pragma hdrstop
#include "DBCal.h"
namespace Dbcal
{
  void __fastcall Register()
  {
    TComponentClass classes[1] = {__classid(TDBCalendar)};
    RegisterComponents("Samples", classes, 0);
  }
}
```

Jelikož tento kalendář bude určen pouze pro čtení, je vhodné znemožnit uživateli provádění změn v ovladači. Provedeme to ve dvou krocích: přidáme vlastnost **ReadOnly** a dovolíme potřebné aktualizace. Když tato vlastnost je nastavena na *true*, jsou všechny buňky v ovladači nevybíratelné. Přidáme deklaraci vlastnosti a soukromou položku k uložení hodnoty:

```
class TDBCalendar : public TSampleCalendar
{
private:
  bool FReadOnly;
protected:
public:
  __fastcall TDBCalendar(TComponent* Owner);
__published:
  __property bool ReadOnly={read=FReadOnly,write=FReadOnly,default=true};
};
```

Zapíšeme definici konstruktoru:

```
__fastcall TDBCalendar::TDBCalendar(TComponent* Owner)
    : TSampleCalendar(Owner)
{
  FReadOnly = true;
}
```

a předefinujeme metodu **SelectCell** k zákazu výběru, jestliže ovladač je pouze pro čtení.

```
bool __fastcall TDBCalendar::SelectCell(long ACol, long ARow)
{
  if (FReadOnly) return false;
  return TSampleCalendar::SelectCell(ACol, ARow);
}
```

Nesmíme zapomenout přidat deklaraci **SelectCell** k deklaraci třídy **TDBCalendar**. Jestliže nyní přidáme kalendář na formulář, zjistíme, že komponenta ignoruje kliknutí a stisky kurzorových kláves. Kalendář pouze pro čtení používá metodu **SelectCell** pro všechny typy změn, včetně nastavování vlastností **Row** a **Col**. Metoda **UdpateCalendar** nastavuje **Row** a **Col** pokaždé, když se změní datum, ale jelikož **SelectCell** nepovolí změny, výběr se nemění, i když se změní datum. K omezení tohoto absolutního zákazu změn, můžeme přidat interní logickou položku ke kalendáři a povolit změny, když je tato položka nastavena na *true*:

```
class TDBCalendar : public TSampleCalendar
{
private:
bool FUpdating;
public:
  virtual void __fastcall UpdateCalendar();
};

bool __fastcall TDBCalendar::SelectCell(long ACol, long ARow)
{
  if (!FUpdating && FReadOnly) return false;
  return TSampleCalendar::SelectCell(ACol, ARow);
}
void __fastcall TDBCalendar::UpdateCalendar()
{
  FUpdating = true;
```

```
  try
  {
    TSampleCalendar::UpdateCalendar();
  }
  catch(...)
  {
    FUpdating = false;
    throw;
  }
  FUpdating = false;
}
```

Kalendář stále neumožňuje uživateli provádět změny data, ale již správně reaguje na změny data provedené změnou vlastností data. Dále potřebujeme ke kalendáři přidat schopnost prohlížet data.

Propojení mezi ovladačem a databází je obsluhováno objektem nazvaným datový spoj. Builder poskytuje několik typů datových spojů. Objekt datového spoje, který propojuje ovladač s jednou položkou v databázi je **TFieldDataLink**. Jsou také datové spoje pro celé tabulky. Objekt závislého ovladače vlastní svůj objekt datového spoje. Tj. ovladač má odpovědnost za vytvoření i uvolnění datového spoje. K vytvoření datového spoje jako vlastněného objektu provedeme tři kroky: deklarujeme objektovou položku, deklarujeme přístupové vlastnosti a inicializujeme datový spoj. Komponenta vyžaduje položku pro každý svůj vlastněný objekt. V našem případě kalendář potřebuje položku typu **TFieldDataLink** pro svůj datový spoj:

```
class TDBCalendar : public TSampleCalendar
{
private:
  TFieldDataLink *FDataLink;
};
```

Dříve než můžeme přeložit aplikaci, musíme vložit hlavičkové soubory *DB.HPP* a *DBTables.HPP* do hlavičkového souboru naší jednotky. Každý datově závislý ovladač má vlastnost **DataSource**, která specifikuje který objekt datového zdroje v aplikaci poskytuje data ovladači. Dále ovladač, který přistupuje k samostatné položce vyžaduje vlastnost **DataField** ke specifikaci položky datového zdroje. Tyto přístupové vlastnosti neposkytují přístup k vlastněnému objektu sami, ale odpovídajícími vlastnostmi ve vlastněném objektu. Deklarujeme vlastnosti **DataSource** a **DataField** a jejich implementační metody a zapíšeme metody jako „předávající“ metody k odpovídajícím vlastnostem objektu datového spojení.

```
class TDBCalendar : public TSampleCalendar
{
private:
  AnsiString __fastcall GetDataField();
  TDataSource *__fastcall GetDataSource();
  void __fastcall SetDataField(const AnsiString Value);
  void __fastcall SetDataSource(TDataSource *Value);
__published:
  __property AnsiString DataField = {read=GetDataField,
                                     write=SetDataField, nodefault};
  __property TDataSource *DataSource = {read=GetDataSource,
                                        write=SetDataSource, nodefault};
};

AnsiString __fastcall TDBCalendar::GetDataField()
{
  return FDataLink->FieldName;
}
TDataSource *__fastcall TDBCalendar::GetDataSource()
{
  return FDataLink->DataSource;
}
void __fastcall TDBCalendar::SetDataField(const AnsiString Value)
{
  FDataLink->FieldName = Value;
}
void __fastcall TDBCalendar::SetDataSource(TDataSource *Value)
{
  FDataLink->DataSource = Value;
}
```

Nyní, když máme vytvořeno spojení mezi kalendářem a jeho datovým spojem, je jeden velmi důležitý krok. Musíme při vytváření ovladače kalendáře vytvořit objekt datového spoje a datový spoj zrušit před zrušením kalendáře. Závislý ovladač vyžaduje přístup k svému datovému spoji prostřednictvím své existence, a musíme tedy vytvořit objekt datového spoje v jeho vlastním konstruktoru a zrušit objekt datového spoje před svým samotným zrušením. Předefinujeme tedy konstruktor a destruktor kalendáře k vytváření a rušení objektu datového spoje.

```
class TDBCalendar : public TSampleCalendar
{
public:
  __fastcall TDBCalendar(TComponent* Owner);
  __fastcall ~TDBCalendar();
};

__fastcall TDBCalendar::TDBCalendar(TComponent* Owner)
    : TSampleCalendar(Owner)
{
  FReadOnly = true;
  FDataLink = new TFieldDataLink();
}
__fastcall TDBCalendar::~TDBCalendar()
{
  delete FDataLink;
}
```

Nyní máme kompletní datový spoj, ale nemáme možnost řídit, která data budou čtena z připojené položky.

Náš ovladač má datový spoj a vlastnosti specifikující datový zdroj a datovou položku a musíme ještě vytvořit reakce na změny v datech této datové položky, neboť se můžeme přesunout na jiný záznam. Všechny objekty datových spojů mají události nazvané **OnDataChange**. Když datový zdroj indikuje změnu ve svých datech, objekt datového spoje volá obsluhu událostí připojenou k této události. K aktualizaci ovladače v reakci na datové změny, připojíme obsluhu k události **OnDataChange** datového spoje. V našem případě, přidáme metodu **DataChange** ke kalendáři a určíme ji jako obsluhu pro **OnDataChange** datového spoje.

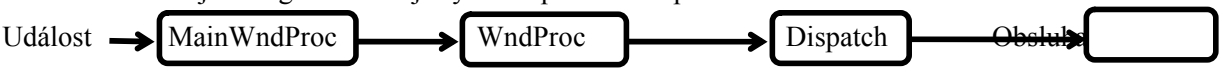
```
class TDBCalendar : public TSampleCalendar
{
private:
  void __fastcall DataChange(TObject *Sender);
};

__fastcall TDBCalendar::TDBCalendar(TComponent* Owner)
    : TSampleCalendar(Owner)
{
  FReadOnly = true;
  FDataLink = new TFieldDataLink();
  FDataLink->OnDataChange = DataChange;
}
__fastcall TDBCalendar::~TDBCalendar()
{
  FDataLink->OnDataChange = NULL;
  delete FDataLink;
}
void __fastcall TDBCalendar::DataChange(TObject *Sender)
{
  if (FDataLink->Field == NULL) CalendarDate = 0;
  else CalendarDate = FDataLink->Field->AsDateTime;
}
```

Tím je zobrazovací databázový ovladač kalendáře hotov. Vytvoření editačního ovladače je mnohem komplikovanější. Dříve než budeme pokračovat ve vývoji této komponenty se seznámíme se zpracováním zpráv Windows.

30. Jedním z klíčů tradičního programování Windows je zpracování zpráv zasílaných z Windows aplikaci. Builder je většinou zpracuje za nás, ale v případě vytváření komponent je možné, že budeme potřebovat zpracovat zprávu, kterou Builder nezpracovává nebo že vytvoříme svou vlastní zprávu a budeme ji potřebovat zpracovat. Všechny objekty Builderu mají mechanismus pro zpracování zpráv. Základní myšlenkou zpracování zpráv je to, že objekt přijme zprávu nějakého typu a zpracuje ji voláním jedné z množiny specifiko-

vaných metod závisejících na přijaté zprávě. Neexistuje-li metoda pro jistou zprávu, je použita implicitní obsluha. Následující diagram ukazuje systém zpracování zpráv:



Knihovna komponent Builderu definuje systém zpracování zpráv, který překládá všechny zprávy Windows (včetně uživatelem definovaných zpráv) určené jistému objektu na volání metod. Tento mechanismus nebudeme nikdy potřebovat měnit. Budeme potřebovat pouze vytvářet metody zpracování zpráv. Zpráva Windows je datový záznam, který obsahuje několik důležitých položek. Nejdůležitější z nich je hodnota, která identifikuje zprávu. Windows definuje mnoho zpráv a soubor *MESSAGES.HPP* deklaruje identifikátory pro všechny z nich. Další důležité informace ve zprávě jsou obsaženy ve dvou položkách parametrů a položce výsledku. Jeden parametr je 16 bitový a druhý 32 bitový. Jak často vidíme v kódu Windows, odkazujeme se na tyto hodnoty jako na *wParam* (word parametr) a *lParam* (long parametr). Často každý z těchto parametrů obsahuje více než jednu informaci a na časti parametrů se odkazujeme makry jako LOWORD a HIWORD. Např. voláním `HIWORD(lParam)` získáme vyšší slovo tohoto parametru. Původně si programátoři Windows museli pamatovat, co každý parametr obsahuje. Později Microsoft tyto parametry pojmenoval, což usnadňuje jejich používání. Např. parametr zprávy WM_KEYDOWN se nyní nazývá *nVirtKey*, což je více informativní než *wParam*.

Builder zjednodušuje systém zpracování zpráv v několika směrech: Každá komponenta dědí kompletní systém zpracování zpráv. Tento systém má implicitní zpracování. Definujeme pouze obsluhy pro zprávy, na které chceme reagovat specificky. Můžeme modifikovat pouze malé části zpracování zpráv a pro většinu zpracování použít zděděné metody. Značnou výhodou tohoto systému zpracování zpráv je, že můžeme bezpečně zasílat kdykoli libovolnou zprávu libovolné komponentě. Jestliže komponenta nemá pro zprávu definovanou obsluhu, pak je použita implicitní obsluha, což obvykle ignoruje zprávu. Builder registruje metodu nazvanou **MainWndProc** jako proceduru okna po každý typ komponenty v aplikaci. **MainWndProc** obsahuje blok zpracování výjimek, předávající záznam zprávy z Windows virtuální metodě nazvané **WndProc** a zpracovávající libovolné výjimky voláním metody **HandleException** objektu aplikace. **MainWndProc** je statická metoda, která neobsahuje speciální zpracování některých zpráv. Přizpůsobení provádíme až v **WndProc**, neboť každý typ komponenty může předefinovat tuto metodu podle svých potřeb. Metody **WndProc** testují zda zpracovávání nemá ignorovat některé neočekávané zprávy. Např. **TWinControl** během tažení komponenty ignorují události klávesnice. **WndProc** předává události klávesnice pouze když komponenta není tažena. Konečně **WndProc** volá **Dispatch**, statickou metodu zděděnou od **TObject**, určující která metoda bude volána k obsluze zprávy. **Dispatch** používá položku **Msg** záznamu zprávy k určení jak zpracovat jistou zprávu. Jestliže komponenta pro zprávu nemá obsluhu **Dispatch** volá **DefaultHandler**.

31. Před změnou zpracování zpráv naší komponenty se ujistíme, že to skutečně musíme udělat. Builder překládá mnoho zpráv Windows na události, které jak tvůrce komponenty, tak i uživatel komponenty může obsloužit. Lépe než měnit chování zpracování zpráv je měnit chování zpracování událostí. Ke změně zpracování zprávy předefinujeme metodu zpracovávající zprávu. Můžeme také zabránit komponentě ve zpracování zprávy při jistých situacích zachycením zprávy. Pro změnu způsobu zpracování jisté zprávy komponentou předefinujeme metodu zpracování zprávy pro tuto zprávu. Jestliže komponenta nemá obsluhu pro jistou zprávu, musíme deklarovat novou metodu obsluhy zprávy. K předefinování metody zpracování zpráv, deklarujeme novou metodu v chránění části naši komponenty a to se stejným jménem jako má metoda kterou předefinováváme a mapujeme metodu na zprávu pomocí tří maker. Tato makra mají tvar:

```
BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(parametr1, parametr2, parametr3)
END_MESSAGE_MAP
```

Parametr1 je index zprávy definovaný Windows, *parametr2* je typ struktury zprávy a *parametr3* je jméno metody zprávy. Mezi BEGIN_MESSAGE_MAP a END_MESSAGE_MAP můžeme vložit několik maker MESSAGE_HANDLER. Např. k předefinování obsluhy zprávy WM_PAINT v komponentě, opětovně deklarujeme metodu **WMPaint** a třemi makry mapujeme metodu na zprávu WM_PAINT:

```
class TMojeKomponenta : public TComponent
{
protected:
  void __fastcall WPPaint(TWMPaint &Message);

BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(WM_PAINT, TWMPaint, WMPaint)
END_MESSAGE_MAP(TComponent)
};
```

Pouze uvnitř metody zpracování zprávy má naše komponenta přístup ke všem parametrům záznamu zprávy. Jelikož zpráva je vždy parametr volaný odkazem, obsluha může změnit v případě potřeby hodnoty parametrů. Často měníme pouze parametr návratové hodnoty zprávy: hodnotu vrácenou voláním **SendMessage**, která zasílá zprávu. Protože se typ parametru **Message** metody zpracování zprávy mění s typem zpracovávané zprávy, je nutné se podívat do dokumentace Windows na jména a význam jednotlivých parametrů. Jestliže se z nějakého důvodu potřebujeme odkazovat na parametr zprávy jejich starým stylem jmen (*wParam*, *lParam* apod.), můžeme přetypovat **Message** na generický typ **TMessage**, který používá tyto jména parametrů. V jistých situacích, může chtít, aby naše komponenta ignorovala jisté zprávy. Tj. můžeme chtít zabránit komponentě od zpracování zprávy svou obsluhou. Zachycení zprávy provedeme předefinováním virtuální metody **WndProc**. Metoda **WndProc** filtruje zprávy před jejich předáním metodě **Dispatch**, která určuje metodu zpracující zprávu. Předefinováním **WndProc**, můžeme změnit filtr zpráv před jejich zpracováním. Předefinování **WndProc** provádíme takto:

```
void __fastcall TMujOvladac::WndProc(TMessage* Message)
{
  // test na určení, zda pokračovat ve zpracování
  TWinControl->WndProc(Message);
}
```

Následuje část metody **WndProc** pro **TControl** jak je implementována ve VCL v Object Pascalu. **TControl** definuje rozsah zpráv myši, které jsou filtrovány, když uživatel provádí tažení. Předefinování **WndProc** tomu pomáhá dvěma způsoby: Můžeme filtrovat interval zpráv namísto specifikování obsluhy pro každou z nich a můžeme zabránit zpracování zpráv v celku a obsluhy nejsou nikdy volány.

```
procedure TControl.WndProc(var Message: TMessage);
begin
  ...
  if (Message.Msg>=WM_MOUSEFIRST)and(Message.Msg<=WM_MOUSELAST) then
    if Dragging then                    {speciální zpracování tažení}
      DragMouseMsg(TWMouse(Message))
    else
      ...                               {normální zpracování}
    end;
  ...                                   {jinak normální proces}
end;
```

32. Přestože Builder poskytuje obsluhy pro mnoho zpráv Windows, můžeme se dostat do situace, kdy budeme potřebovat vytvořit novou obsluhu zpráv a to když definujeme svou vlastní zprávu. Práce s uživatelskými zprávami má dva aspekty: definování své vlastní zprávy a deklarování nové metody zpracování zprávy. Několik standardních komponent definuje zprávy pro interní použití. Smyslem pro definování zpráv je vysílání informací nepodporované standardními zprávami Windows a oznámení změny stavu. Definování zprávy je dvoukrokový proces: deklarujeme identifikátor zprávy a deklarujeme typ záznamu zprávy. Identifikátor zprávy je celočíselná konstanta. Windows rezervuje zprávy do 1024 pro svoje vlastní použití a tak když deklarujeme svou vlastní zprávu musíme začít nad touto úrovní. Konstanta WM_USER reprezentuje počáteční číslo pro uživatelem definované zprávy. Když definujeme identifikátory zpráv, musíme začít od WM_USER. Musíme si být vědomi, že některé standardní ovladače Windows používají zprávy v rozsahu uživatelských definic. Jsou to seznamy, kombinovaná okna, editační okna a tlačítka. Jestliže odvozujeme komponentu od některé z nich a chceme definovat novou zprávu pro ní, je potřeba se podívat do souboru *MESSAGES.HPP* a zjistit, které zprávy Windows jsou skutečně definované pro tyto ovladače. Následující kód ukazuje dvě uživatelem definované zprávy:

```
#define WM_MOJEPRVNIZPRAVA (WM_USER + 400)
#define WM_MOJEDRUHAZPRAVA (WM_USER + 401)
```

Jestliže chceme dát smysluplná jména parametrům naší zprávy, je potřeba deklarovat typ struktury zprávy pro tuto zprávu. Struktura zprávy je typ předávaného parametru metodě zpracování zprávy. Jestliže nepoužíváme parametr zprávy nebo jestliže chceme použít starý způsob zápisu parametrů (*wParam*, *lParam* apod.), můžeme použít implicitní záznam zprávy **TMessage**. Při deklaraci typu struktury zprávy používáme tyto konvence: Jméno typu záznamu vždy začíná **T** a následuje jméno zprávy. Jméno první položky v záznamu je **Msg** a je typu **Cardinal**. Definujeme další dvě slabiky, které odpovídají *wParam*. Definujeme další čtyři slabiky, které odpovídají *lParam*. Nakonec přidáme položku nazvanou *Result*, která je typu *Longint*. Následuje záznam zpráv pro všechny zprávy myši, **TWMMouse**:

```
struct TWMMouse {
    Cardinal Msg;
    long Keys;
    union
```

```
    {
          struct
          {
                Windows::TSmallPoint Pos;
                long Result;
          };
          struct
          {
                short XPos;
                short YPos;
          };
    };
};
```

Jsou dvě situace, které vyžadují deklarování nové metody zpracování zprávy: naše komponenta vyžaduje zpracování zprávy Windows, která není zpracovávána standardními komponentami a definování své vlastní zprávy pro použití v našich komponentách. Deklaraci metody zpracování zprávy provedeme takto: Deklarujeme metodu v chráněné části deklarace třídy komponenty. Ujistíme se, že metoda vrací void. Nazveme metodu po zpracovávané zprávě, ale bez znaků podtržení. Předáváme jeden parametr volaný odkazem nazvaný *Message*, typu struktury zprávy. Mapujeme metodu na zprávu použitím maker. Zapíšeme kód pro specifické zpracování v komponentě. Voláme zděděnou obsluhy zprávy. Následuje deklarace obsluhy zprávy pro uživatelem definovanou zprávu nazvanou CM_CHANGECOLOR:

```
#define CM_CHANGECOLOR (WM_USER + 400)
class TMojeKomponenta : public TControl
{
protected:
  void __fastcall CMChangeColor(TMessage &Message);
BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(CM_CHANGECOLOR, TMessage, CMChangeColor)
END_MESSAGE_MAP(TComtrol)

void __fastcall TMojeKomponenta::CMChangeColor(TMessage &Message)
{
  Color = Message->LParam;
  TControl::CMChangeColor(Message);
}
```

33. Budeme pokračovat ve vývoji předchozí komponenty a umožníme editovat připojenou datovou položku. Protože se jedná o editační ovladač musíme implicitně nastavit jeho vlastnost **ReadOnly** na *false* (v konstruktoru i v definici vlastnosti). Nás ovladač musí reagovat na zprávy Windows týkající se myši (WM_LBUTTONDOWN, WM_MBUTTONDOWN a WM_RBUTTONDOWN) a zprávy klávesnice (WM_KEYDOWN). K umožnění, aby ovladač reagoval na tyto zprávy, musíme zapsat obsluhy reagující na tyto zprávy. Chráněná metoda **MouseDown** je metoda pro událost ovladače **OnMouseDown**. Ovladač sám volá **MouseDown** v reakci na zprávu stisknutí tlačítka myši od Windows. Když předefinováváme zděděnou metodu **MouseDown**, můžeme vložit kód, který poskytuje ostatní reakce voláním události **OnMouseDown**. Do třídy **TDBCalendar** přidáme metodu **MouseDown**:

```
class TDBCalendar : public TSampleCalendar
{
protected:
  virtual void __fastcall MouseDown(TMouseButton Button,
                                    TShiftState Shift, int X, int Y);
};
```

a do souboru CPP tuto metodu zapíšeme:

```
void __fastcall TDBCalendar::MouseDown(TMouseButton Button,
                                       TShiftState Shift, int X, int Y)
{
  TMouseEvent MyMouseDown;
  if (!FReadOnly && FDataLink->Edit())
    TSampleCalendar::MouseDown(Button, Shift, X, Y);
  else {
    MyMouseDown = OnMouseDown;
    if (MyMouseDown != NULL) MyMouseDown(this, Button, Shift, X, Y);
  }
}
```

Když **MouseDown** reaguje na zprávu myši, pak zděděná metoda **MouseDown** je volána pouze, jestliže vlastnost **ReadOnly** ovladače je nastavena na *false* a jestliže objekt datového spoje je v editačním režimu (položka může být editována). Jestliže položka nemůže být editována, pak je provedena obsluha události **OnMouseDown** (existuje-li).

Metoda **KeyDown** je chráněná metoda pro událost **OnKeyDown** ovladače. Ovladač sám volá **KeyDown** v reakci na zprávu stisknutí klávesy od Windows. Obdobně jako **MouseDown** předefinujeme i **KeyDown**:

```
class TDBCalendar : public TSampleCalendar
{
protected:
  virtual void __fastcall KeyDown(unsigned short &Key,TShiftState Shift);
};

void __fastcall TDBCalendar::KeyDown(unsigned short &Key,TShiftState Shift)
{
  TKeyEvent MyKeyDown;
  Set<unsigned short, 0, 8> keySet;
  keySet = keySet << VK_UP << VK_DOWN << VK_LEFT << VK_RIGHT << VK_END <<
                     VK_HOME << VK_PRIOR << VK_NEXT;
  if (!FReadOnly && (keySet.Contains(Key)) && FDataLink->Edit())
    TCustomGrid::KeyDown(Key, Shift);
  else {
    MyKeyDown = OnKeyDown;
    if (MyKeyDown != NULL) MyKeyDown(this, Key, Shift);
  }
}
```

Jsou dva typy datových změn: změna v hodnotě položky, která musí být zohledněna v ovladači a změna v ovladači, která musí být provedena v datové položce. Komponenta **TDBCalendar** má metodu **DataChange**, která zpracovává změny v hodnotě položky a tak první typ změn je již ošetřen. Třída položky datového spoje má událost **OnUpdateData**, která nastane, když uživatel modifikuje obsah ovladače. Ovladač kalendáře má metodu **UpdateData**, kterou můžeme použít k obsloužení této události. Přidáme tedy metodu **UpdateData** do deklarace třídy formuláře:

```
class TDBCalendar : public TSampleCalendar
{
private:
  void __fastcall UpdateData(TObject *Sender);
};
```

do souboru CPP zapíšeme metodu **UpdateData**

```
void __fastcall TDBCalendar::UpdateData(TObject *Sender)
{
  FDataLink->Field->AsDateTime = CalendarDate;
}
```

a v konstruktoru **TDBCalendar** přiřadíme metodu **UpdateData** události **OnUpdateData**

```
__fastcall TDBCalendar::TDBCalendar(TComponent* Owner)
    : TSampleCalendar(Owner)
{
  FReadOnly = false;
  FDataLink = new TFieldDataLink();
  FDataLink->OnDataChange = DataChange;
  FDataLink->OnUpdateData = UpdateData;
}
```

Když je nastavena nová hodnota, pak je volána metoda **Change** ovladače kalendáře. **Change** volá obsluhu události **OnChange** (pokud existuje). Uživatel komponenty může zapsat kód obsluhy události **OnChange** k reagování na změnu datumu. Musíme tedy přidat novou metodu **Change** do komponenty **TDBCalendar**

```
class TDBCalendar : public TSampleCalendar
{
protected:
  virtual void __fastcall Change();
};
```

a zapsat metodu **Change**, volající metodu **Modified**, která informuje datový spoj, že datum bylo změněno a potom volá zděděnou metodu **Change**

```
void __fastcall TDBCalendar::Change()
```

```
{
  if (FDataLink != NULL) FDataLink->Modified();
  TSampleCalendar::Change();
}
```

Posledním krokem při vytváření editovatelného ovladače je aktualizace datového spoje na novou hodnotu. To nastává, když změníme hodnotu v ovladači a ovladač opustíme (klinutím mimo ovladač nebo stiskem klávesy Tab). VCL má definovány zprávy pro operace s ovladačem. Např. zpráva CM_EXIT je zaslána, když uživatel ovladač opustí. Můžeme zapsat obsluhu zprávy, která na zprávu bude reagovat. V našem případě, když uživatel opustí ovladač, metoda **CMExit** (obsluha zprávy pro CM_EXIT) reaguje aktualizací záznamu v datovém spoji na změněnou hodnotu. Do komponenty přidáme obsluhu zprávy

```
class TDBCalendar : public TSampleCalendar
{
private:
  void __fastcall CMExit(TWMNoParams &Message);

BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(CM_EXIT, TWMNoParams, CMExit)
END_MESSAGE_MAP(TSampleCalendar)
};
```

a do souboru CPP zapíšeme:

```
void __fastcall TDBCalendar::CMExit(TWMNoParams & Message)
{
  try
  {
    FDataLink->UpdateRecord();
  }
  catch(...)
  {
    SetFocus();
    throw;
  }
}
```

Tím je vývoj editovatelné komponenty hotov.

34. Často používané komponenty dialogových oken můžeme také přidat na Paletu komponent. Naše komponenta dialogového okna bude pracovat stejně jako komponenty které reprezentují standardní dialogová okna Windows. Vytvoření komponenty dialogového okna vyžaduje čtyři kroky: definování rozhraní komponenty, vytvoření a registraci komponenty, vytvoření rozhraní komponenty a testování komponenty. Cílem je vytvořit jednoduchou komponentu, kterou uživatel může přidat k projektu a nastavit její vlastnosti během návrhu. „Obalovací“ komponenta Builderu přiřazená k dialogovému oknu je vytvoří a provede při běhu aplikace a předá data definovaná uživatelem. Komponenta dialogového okna je tedy zase opětovně použitelná a přizpůsobitelná. Dále si ukážeme jak vytvořit obalovou komponentu okolo generického formuláře **About** obsaženého v Galerii Builderu. Nejprve zkopírujeme soubory ABOUT.H, ABOUT.CPP a ABOUT.DFM do našeho pracovního adresáře. ABOUT.CPP vložíme do nějaké aplikace a provedeme překlad. Tím vytvoříme soubor ABOUT.OBJ, který budeme potřebovat při vytváření komponenty.

    Dříve než můžeme vytvořit komponentu pro naše dialogové okno, musíme určit jak chceme, aby ji vývojář používal. Vytvoříme rozhraní mezi našim dialogovým oknem a aplikací, která jej používá. Např. podívejme se na vlastnosti komponenty společného dialogového okna. Umožňují vývojáři nastavovat počáteční stav dialogového okna, jako je titulek a počáteční nastavení ovladačů, a po uzavření dialogového okna převzít zpět požadované informace. Není to přímá interakce s jednotlivými ovladači v dialogovém okně, ale s vlastnostmi v obalové komponentě. Rozhraní tedy musí obsahovat požadované informace, které formulář dialogového okna může zobrazit a vracet aplikaci. Můžeme si představit vlastnosti obalové komponenty jako data přenášená z a do dialogového okna. V případě okna **About**, nepotřebujeme vracet žádné informace a tedy vlastnosti obalové komponenty obsahují pouze informace požadované k zobrazení v okně. Jsou to čtyři položky dialogového okna **About**, které aplikace může ovlivnit a poskytneme tedy čtyři vlastnosti typu řetězce.

    Obvyklým způsobem vytvoříme komponentu. Zadáme tato specifika: programovou jednotku komponenty nazveme *AboutDlg*, od **TComponent** odvodíme nový typ komponenty **TAboutBoxDlg** a registrujeme vytvářenou komponentu na stránce *Samples* Palety komponent. Po provedení těchto akcí dostaneme:

```
#ifndef AboutDlgH
#define AboutDlgH
#include <vcl\SysUtils.hpp>
```

```
#include <vcl\Controls.hpp>
#include <vcl\Classes.hpp>
#include <vcl\Forms.hpp>
class TAboutBoxDlg : public TComponent
{
private:
protected:
public:
  __fastcall TAboutBoxDlg(TComponent* Owner);
__published:
};
#endif

#include <vcl\vcl.h>
#pragma hdrstop
#include "AboutDlg.h"
__fastcall TAboutBoxDlg::TAboutBoxDlg(TComponent* Owner)
    : TComponent(Owner)
{
}
namespace Aboutdlg
{
    void __fastcall Register()
    {
          TComponentClass classes[1] = {__classid(TAboutBoxDlg)};
          RegisterComponents("Samples", classes, 0);
    }
}
```

Nyní, když máme vytvořenou komponentu a definované rozhraní mezi komponentou a dialogovým oknem, můžeme implementovat její rozhraní. To provedeme ve třech krocích: vložíme jednotku formuláře, přidáme vlastnosti rozhraní a přidáme metodu **Execute**. Pro naší obalovou komponentu k inicializaci a zobrazení obaleného dialogového okna musíme přidat jednotku formuláře okna do jednotky obalové komponenty. Vložíme tedy *About.h* a sestavení s ABOUT.OBJ do hlavičkového souboru komponenty:

```
#include "About.h"
#pragma link "About.obj"
```

Jednotka formuláře vždy deklaruje instanci typu formuláře. V případě okna **About**, je typ formuláře **TAboutBox** a soubor *About.h* obsahuje následující deklaraci:

```
extern TAboutBox *AboutBox;
```

Vlastnosti v obalové komponentě jsou jednodušší než vlastnosti v normální komponentě. Umožňují pouze předávání dat mezi obalovou komponentou a dialogovým oknem. Vložením dat do vlastností formuláře, povolíme vývojáři nastavit data během návrhu pro obal k jejich předání do dialogového okna při běhu aplikace. Deklarace vlastnosti rozhraní vyžaduje dvě další deklarace v typu komponenty: soukromou položku, kterou obal použije k uložení hodnoty vlastnosti a samotnou zveřejňovanou deklaraci vlastnosti, která specifikuje jméno vlastnosti a říká která položka je použita pro uložení. Vlastnosti rozhraní nevyžadují přístupové metody. Používají přímý přístup ke svým datům. Podle konvencí má objektová položka pro uložení hodnoty vlastnosti stejné jméno jako vlastnost, ale na začátku je přidáno písmeno **F**. Např. k deklaraci vlastnosti rozhraní celočíselného typu nazvané *Rok*, použijeme:

```
class TMujObal : public TComponent
{
private:
  int FRok;
__published:
  __property int Rok = {read=FRok, write=FRok};
}
```

Pro dialogové okno **About** potřebujeme čtyři vlastnosti typu String, po jedné pro jméno produktu, informaci o verzi, autorských právech a komentář:

```
class TAboutBoxDlg : public TComponent
{
private:
  String FProductName, FVersion, FCopyright, FComments;
__published:
  __property String ProductName={read=FProductName, write=FProductName};
```

```
  __property String Version = {read=FVersion, write=FVersion};
  __property String Copyright = {read=FCopyright, write=FCopyright};
  __property String Comments = {read=FComments, write=FComments};
};
```

Když nyní instalujeme komponentu na paletu a umístíme ji na formulář, můžeme nastavovat vlastnosti a tyto hodnoty jsou automaticky předávány s formulářem. Tyto hodnoty jsou pak použity při provádění dialogového okna. Poslední částí rozhraní komponenty je cesta k otevření dialogového okna a vrácení výsledku při jeho uzavření. Komponenty společných dialogových oken používají logickou funkci nazvanou **Execute**, která vrací *true*, jestliže uživatel stiskl OK nebo *false*, když uživatel okno zrušil. Deklarace pro metodu **Execute** je vždy tato:

```
class TMujObal : public TComponent
{
public:
  bool __fastcall Execute();
}
```

Minimální implementace pro **Execute** vyžaduje vytvoření formuláře dialogového okna, jeho zobrazení jako modálního dialogového okna a vrácení *true* nebo *false* (v závislosti na návratové hodnotě **ShowModal**). Následuje minimální metoda **Execute** pro formulář dialogového okna typu **TMojeDialOkno**:

```
bool __fastcall TMujObal::Execute()
{
  bool Result;
  DialOkno = new TMojeDialOkno(Application);
  Try
  {
    Result = (DialOkno->ShowModal() == IDOK)
  }
  catch(...)
  {
    Result = false;
  }
  DialOkno->Free();
}
```

V praxi, bývá více kódu uvnitř bloku výjimky. Před voláním **ShowModal**, obal nastaví nějaké vlastnosti dialogového okna na základě vlastností rozhraní obalové komponenty. Po návratu z **ShowModal**, obal pravděpodobně nastaví některé své vlastnosti rozhraní na základě provedení dialogového okna. V případě okna **About**, použije obalová komponenta čtyři vlastnosti rozhraní k nastavení obsahu formuláře dialogového okna **About**. Jelikož okno **About** nevrací žádné informace, není nutno provádět nic po volání **ShowModal**. Naše metoda **Execute** bude tedy vypadat takto:

```
bool __fastcall TAboutBoxDlg::Execute()
{
  bool Result;
  AboutBox = new TAboutBox(Application);
  try
  {
    if (ProductName == "") ProductName = Application->Title;
    AboutBox->ProductName->Caption = ProductName;
    AboutBox->Version->Caption = Version;
    AboutBox->Copyright->Caption = Copyright;
    AboutBox->Comments->Caption = Comments;
    AboutBox->Caption = "About "+ ProductName;
    Result = (AboutBox->ShowModal() == IDOK);
  }
  catch(...)
  {
    Result = false;
  }
  AboutBox->Free();
  return Result;
}
```

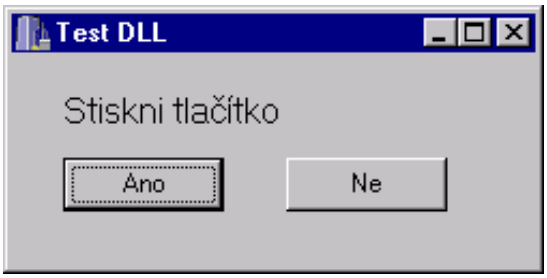
Když instalujeme komponentu dialogového okna, můžeme ji používat stejně jako společná dialogová okna, umístěním na formulář a jejich provedením. Rychlý způsob k otestování okna **About** je přidat na formulář tlačítko a provést dialogové okna při stisknutí tohoto tlačítka. Např. jestliže vytvoříme dialogové okno, uděe-

láme jeho komponentu a přidáme ji na Paletu komponent, pak můžeme testování provést v těchto krocích: vytvoříme nový projekt, umístíme komponentu okna **About** a komponentu tlačítka na formulář, dvojitě klikneme na tlačítko (vytvoříme prázdnou obsluhu události stisku tlačítka), do obsluhy události stisku tlačítka zapíšeme následující řádek kódu:

```
AboutBoxDlg1->Execute();
```

a spustíme aplikaci. Můžeme také vyzkoušet nastavit různé vlastnosti komponenty.

35. Když chceme používat stejné dialogové okno v mnoha aplikacích, obzvláště když všechny aplikace nejsou aplikacemi Builderu, můžeme vytvořit dialogové okno v DLL. Protože DLL je samostatný proveditelný soubor, aplikace zapsané v jiných nástrojích než Builderu, mohou volat stejné DLL. Např. můžeme DLL volat z aplikací vytvořených v C++, Delphi, Paradoxu nebo dBASE. Protože DLL je standardní soubor, každá DLL obsahuje hlavičku knihovny komponent (okolo 100K). Můžeme minimalizovat tuto hlavičku vložením několika dialogových oken do jedné DLL. Vytvoření dialogového okna v DLL vyžaduje tři kroky: přidáme



funkci rozhraní, modifikujeme projektový soubor a otevřeme dialogové okno z aplikace. Předpokládejme, že chceme vytvořit DLL zobrazující následující jednoduché dialogové okno:

Kód pro DLL dialogového okna je tento:

```
// DLLMAIN.H
#ifndef dllMainH
#define dllMainH
#include <vcl\Classes.hpp>
#include <vcl\Controls.hpp>
#include <vcl\StdCtrls.hpp>
#include <vcl\Forms.hpp>
class TYesNoDialog : public TForm
{
__published:    // IDE-managed Components
    TLabel *LabelText;
    TButton *YesButton;
    TButton *NoButton;
    void __fastcall YesButtonClick(TObject *Sender);
    void __fastcall NoButtonClick(TObject *Sender);
private:        // User declarations
    bool returnValue;
public:         // User declarations
    virtual __fastcall TYesNoDialog(TComponent* Owner);
    bool __fastcall GetReturnValue();
};
// exportování funkce rozhraní
extern "C" __declspec(dllexport) bool InvokeYesNoDialog();
extern TYesNoDialog *YesNoDialog;
#endif

// DLLMAIN.CPP
#include <vcl\vcl.h>
#pragma hdrstop
#include "dllMain.h"
#pragma resource "*.dfm"
TYesNoDialog *YesNoDialog;
__fastcall TYesNoDialog::TYesNoDialog(TComponent* Owner)
  : TForm(Owner)
{
   returnValue = false;
}
void __fastcall TYesNoDialog::YesButtonClick(TObject *Sender)
{
```

```
    returnValue = true;
    Close();
}
void __fastcall TYesNoDialog::NoButtonClick(TObject *Sender)
{
    returnValue = false;
    Close();
}
bool __fastcall TYesNoDialog::GetReturnValue()
{
    return returnValue;
}
// exportování standardní C++ funkce rozhraní, kterou voláme mimo VCL
bool InvokeYesNoDialog()
{
    bool returnValue;
    TYesNoDialog *YesNoDialog = new TYesNoDialog(NULL);
    YesNoDialog->ShowModal();
          returnValue = YesNoDialog->GetReturnValue();
          delete YesNoDialog;
    return returnValue;
}
```

Kód v tomto příkladě zobrazuje dialogové okno a ukládá hodnotu určující stisknuté tlačítko do soukromé položky **returnValue**. Tuto hodnotu můžeme získat veřejnou funkcí **GetReturnValue**. K zobrazení dialogového okna a určení které tlačítko bylo stisknuto volá aplikace exportovanou funkci **InvokeYesNoDialog**. Tato funkce je deklarována v DLLMAIN.H jako exportovaná funkce C (k zabránění komolení jmen C++) a používající standardní volací konvence C. Funkce je definována v DLLMAIN.CPP. To umožňuje aby tato funkce umístěná v DLL byla volána libovolnou aplikací (nejen aplikací vytvořenou v Builderu). Po vytvoření funkce rozhraní pro dialogové okno, musíme ještě modifikovat projektový soubor, aby vytvořil DLL namísto aplikace. K přeložení a sestavení DLL z IDE Builderu, nastavíme volbu **Application Target** na stránce **Linker** okna **Project Option** na **Generate DLL** a normálně projekt přeložíme.

Po dokončení zabalení našeho dialogového okna do DLL a překladu DLL, můžeme již používat dialogové okno z aplikací. K použití dialogového okna z DLL provést dvě věci: importovat funkci rozhraní z DLL a volat funkci rozhraní.