Borland C++Builder 5. Ćwiczenia praktyczne

Andrzej Daniluk

Spis treści

Rozdział 1. Pierwsze spotkanie ze środowiskiem Borland C++Builder 5 5
C++ Builder Enterprise 5
C++ Builder Profesional 5
C++ Builder Standard 5
Parę pożytecznych skrótów nazw 6
Technologia OLE 6
OLE Automation 6
Model COM 6
Technologia ActiveX 6
Środowisko programisty — IDE 6
Struktura głównego menu 8
Menu File 8
Menu Edit 11
Menu Search 13
Menu View 14
Menu Project 16
Menu Run 18
Menu Component 20
Menu Tools 21
Menu Help 22
Menu Desktop 23
Pasek narzędzi — Speed Bar 24
Inspektor obiektów — Object Inspector 24
Karta właściwości — Properties 24
Karta obsługi zdarzeń — Events 25
Podsumowanie 26

Rozdział 2. Borland C++Builder 5. Pierwsze kroki 27
Ogólna postać programu pisanego w C++ 27
Funkcja main() 28
Dyrektywa #include i prekompilacja 29
Dyrektywa #pragma hdrstop 30
Dyrektywa #pragma argsused 30
Konsolidacja 30
Konfigurujemy Opcje Projektu 30
Uruchamiamy program 33
Podsumowanie 34

Rozdział 3. Elementarz C++ 35
Podstawowe typy danych oraz operatory arytmetyczne 35
Ćwiczenia do samodzielnego wykonania 36
Operatory relacyjne i logiczne 37
Deklarowanie tablic 37
Instrukcje sterujące 38
Instrukcja if 39
Ćwiczenie do samodzielnego wykonania 39
Instrukcja switch 39
Ćwiczenie do samodzielnego wykonania 40
Instrukcja for 41
Ćwiczenie do samodzielnego wykonania 42
Nieskończona pętla for 42
Instrukcja while 42
Ćwiczenie do samodzielnego wykonania 43
Instrukcja do. . .while 43
Ćwiczenie do samodzielnego wykonania 44

Funkcje w C++ ... 44
Ćwiczenie do samodzielnego wykonania ... 45
Wskazania i adresy ... 45
Struktury ... 46
Ćwiczenie do samodzielnego wykonania ... 48
Podsumowanie ... 48

Rozdział 4. Projektowanie obiektowe OOD ... 49
Klasa ... 49
Obiekt ... 49
Metody ... 49
Widoczność obiektów ... 49
Współdziałanie obiektów ... 50
Implementacja obiektu ... 50
Zdarzenie ... 50
Dziedziczenie ... 50
Programowanie zorientowane obiektowo ... 50
Klasa TForm1 ... 51
Konstruktor TForm1() ... 51
Formularz jako zmienna obiektowa ... 52
Tworzymy aplikację ... 52
Pierwsza aplikacja ... 52
Funkcja obsługi zdarzenia ... 54
Ogólna postać aplikacji w C++Builder 5 ... 58
Wykorzystujemy własną strukturę ... 59
Ćwiczenie do samodzielnego wykonania ... 61
Wykorzystujemy własną funkcję ... 61
Ćwiczenie do samodzielnego wykonania ... 63
Podsumowanie ... 64

Rozdział 5. Podstawowe elementy biblioteki VCL ... 65
Hierarchia komponentów VCL ... 65
Klasa TObject ... 66
Klasa TPersistent ... 66
Klasa TComponent ... 66
Klasa TControl ... 66
Właściwości klasy TControl ... 66
Zdarzenia klasy TControl ... 67
Klasa TGraphicControl ... 69
Klasa TWinControl ... 70
Metody klasy TWinControl ... 70
Właściwości klasy TWinControl ... 70
Zdarzenia klasy TWinControl ... 70
Podsumowanie ... 71

Rozdział 6. Biblioteka VCL ... 72
Karta Standard ... 72
TFrames ... 73
Zastosowanie TFrames ... 74
Wykorzystanie pozostałych komponentów karty Standard ... 77
Wczytujemy plik z dysku ... 78
Komponenty TRadioGroup oraz TScrollBar ... 79
Komponenty TMainMenu oraz TPopupMenu ... 80
TPanel oraz TCheckBox ... 81
Przykładowa aplikacja ... 82
Ćwiczenie do samodzielnego wykonania ... 84
Hierarchia własności obiektów Właściciele i rodzice ... 84
Ćwiczenie do samodzielnego wykonania ... 85
Karta Additional ... 85
Karta Win32 ... 86
Karta System ... 87
Karta Dialogs ... 88

Tworzymy profesjonalne menu 88
Przykład wykorzystania komponentów TApplicationEvents oraz TTimer 95
Ćwiczenie do samodzielnego wykonania 97
Karta Win 3.1 97
Wykorzystanie komponentów TDirectoryListBox, TFileListBox, TFilterComboBox oraz TDriveComboBox 98
Karta Samples 99
Wykorzystanie komponentów TCSpinEdit, TTrayIcon, TImageList oraz TCheckBox100
Komponent TCCalendar 102
Karta ActiveX 103
Komponent TF1Book 104
Karta Internet 106
Karta Servers 107
Komponenty TPowerPointApplication, TWordApplication oraz TExcelApplication107
Podsumowanie 109

Rozdział 7. **Aplikacje SDI oraz MDI 110**

Aplikacje jednodokumentowe 110
Aplikacje wielodokumentowe 111
Podsumowanie 112

Wprowadzenie

Jeden z najnowszych produktów firmy Borland/Imprise — C++Builder 5 — reprezentuje niezwykle bogate i bardzo wydajne środowisko programistyczne. Zapoznanie się z nowym Builderem może stanowić pretekst do pokazania Czytelnikom pewnych elementów współczesnych metod programowania aplikacji. W zamierzeniu książka ta przeznaczona jest dla osób dopiero zaczynających przygodę z programowaniem obiektowym. W jej trakcie będziemy stopniowo poznawać niezwykle bogate środowisko programistyczne oferowane nam przez Buildera 5. Jednocześnie zapoznamy się z najbardziej podstawowymi elementami oraz metodami konstrukcji algorytmów właściwymi dla Borland C++, tak by w efekcie w pewnym momencie uświadomić sobie, że oto zaczynamy samodzielnie tworzyć aplikacje przy pomocy Borland C++Buildera 5 jako całości. Poznamy strukturę programów pisanych zarówno w C++, jak i C++Builderze, a także zaznajomimy się z pojęciem klasy oraz obiektu formularza. Ważnym celem książki jest zaciekawienie Czytelnika i zachęcenie go do przyjęcia postawy eksploracyjnej, niezbędnej we współczesnym świecie.

Nie będzie naszym zadaniem przedstawienie skomplikowanych technik związanych z algorytmizacją programów oraz stosowaniem wyszukanych funkcji, struktur czy innych obiektów tak charakterystycznych dla współczesnego C++. Skoncentrujemy się natomiast na poznaniu środowiska programisty oferowanego przez C++Buildera 5 wraz z podstawowymi elementami biblioteki VCL. Główny nacisk zostanie położony na umiejętność wykorzystania już istniejących obiektów, tak aby nawet zaczynający swą przygodę ze współczesnym C++ Czytelnik nie czuł się zagubiony w gąszczu skomplikowanych terminów i aby w trakcie całej książki miał wyraźny przegląd sytuacji. Wykonując proste ćwiczenia nauczymy się posługiwać właściwościami, zdarzeniami oraz metodami różnych komponentów. Zamieszczone w książce przykłady kompletnych aplikacji pomogą nam zrozumieć, jak z prezentowanych komponentów możemy skorzystać w praktyce. Książka ta nie zakłada znajomości wcześniejszych wersji Buildera, dlatego oprócz elementów biblioteki VCL, właściwych dla nowego Buildera, omówimy też sposoby korzystania z zasobów zaadaptowanych ze starszych jego wersji, o których pliki pomocy wyrażają się w sposób bardzo oszczędny.

Komentarz: Zmieniłabym na: „C++Builder 5, jeden z najnowszych produktów firmy Borland/Imprise, reprezentuje niezwykłe możliwości programistyczne.

Usunięto: też

Usunięto: ,

Komentarz: Czy chodzi o: „niezwykle duży wybór metod programowania oferowany [...]"?

Usunięto: j

Usunięto: tak

Usunięto: Ś

Komentarz: Czy to poprawne sformułowanie?

Usunięto: z tego powodu,

Rozdział 1 Pierwsze spotkanie ze środowiskiem Borland C++Builder 5

Najważniejszym elementem nowego Buildera jest szybki optymalizujący kompilator Borland C++ Compiler v. 5.5. C++, zgodnie ze wszystkimi liczącymi się wersjami standardu ANSI/ISO, sprawia, że praca z C++ Builderem staje się jeszcze łatwiejsza. C++ Builder dostępny jest w trzech wersjach różniących się stopniem zaawansowania.

C++ Builder Enterprise

Głównym jego zastosowaniem jest tworzenie aplikacji rozproszonych, internetowych oraz typu klient/serwer. Wbudowane komponenty Internet Express, zawierające kreatory klientów internetowych, bardzo ułatwiają tworzenie w pełni skalowalnych aplikacji, zdolnych dynamicznie przesyłać dane poprzez WWW. Programista ma do dyspozycji języki HTML 4 i XML. Tworzenie aplikacji rozproszonych ułatwiają MIDAS, PageProducer oraz WebBroker. ADOExpress zapewnia bardzo szybki dostęp do danych praktycznie rzecz biorąc z dowolnych źródeł. Tworzone w ten sposób aplikacje będą działać na różnych platformach internetowych. Większa wydajność pracy grup programistów została zapewniona przez TeamSource. Mamy tutaj możliwości grupowania projektów wraz z ich jednoczesną kompilacją; możliwa jest również kompilacja w tle.

C++ Builder Profesional

Posługując się tą wersją mamy możliwość szybkiego tworzenia aplikacji sieciowych poprzez wbudowane biblioteki elementów internetowych oraz perfekcyjnie zorganizowaną obsługę baz danych. Posługując się technologią CodeGuard można zminimalizować występowanie różnego rodzaju błędów alokacji i dostępu do pamięci. Wykorzystanie komponentów Frame pozwala na efektywne, wizualne tworzenie komponentów biznesowych. Budowanie aplikacji posługującej się relacyjnymi bazami danych ułatwia InterBase Express.

C++ Builder Standard

Rozpoczęcie pracy min jest najlepszym i najprostszym sposobem poznania C++ oraz nauczenia się metod wizualnego budowania aplikacji. Do dyspozycji mamy kilkadziesiąt komponentów wizualnych oferowanych przez biblioteki VCL (ang. *Visual Component Library*). Wersja Standard udostępnia nam wszystkie niezbędne zasoby interfejsu programisty Win32 API (ang. *Application Programming Interface*). Dzięki niej mamy możliwość wykorzystywania zaawansowanych technologii obiektowych, takich jak COM czy ActiveX. Z kolei OLE Automation umożliwia naszym aplikacjom współpracę z elementami pakietu MS Office, np. Word, Excel, Power Point czy Outlook.

Komentarz: Czy to prawidłowe określenie?

Usunięto: Będąc zgodnym

Usunięto: C++

Usunięto: ła

Usunięto: Tradycyjnie

Komentarz: ?

Usunięto: internetowych

Usunięto: ,

Usunięto: ,

Usunięto: t

Usunięto: ,

Komentarz: Bardzo nieczytelna charakterystyka języka C++Builder Enterprise (bo chyba rzecz dotyczy języka programowania?). W końcówce tego akapitu już zupełnie nie wiadomo o co chodzi. Co to jest TeamSource i czy to TeamSource stwarza możliwości grupowania projektów (jakich projektów), czy też takie możliwości ma C++Builder Enterprise?

Komentarz: Komponenty biznesowe — cóż to takiego?

Komentarz: ?

Komentarz: „wizualnego budowania” czy „budowania wizualnych aplikacji”?

Komentarz: Może „elementów, części, składników”? Słowo „komponent” ciągle się tutaj powtarza.

Komentarz: Czy chodzi o interfejs programu Win32 API? „Programista” to osoba zajmująca się programowaniem, tworzeniem programu, tymczasem mam wrażenie, że Autor używa tego słowa w znaczeniu „program, programowanie”.

Komentarz: Czy to prawidłowy zapis? Czy może: Win.3.2?

Usunięto: M

Komentarz: Jak się ma OLE Automation do C++Builder Standard?

Parę pożytecznych skrótów nazw

Ponieważ zaczynamy właśnie swoją przygodę z programowaniem obiektowym, pożytecznym będzie, jeżeli zapoznamy się z paroma najczęściej używanymi skrótami pewnych nazw, z którymi możemy się spotkać czytając różnego rodzaju artykuły. Bardzo często nazwy takie pojawiają się też w plikach pomocy udostępnianych przez C++ Builder 5.

Technologia OLE

OLE (ang. *Object Linking and Embedding*) umożliwia osadzanie, łączenie i wzajemną wymianę różnych obiektów danych przy jednoczesnej pracy wielu aplikacji Windows (OLE 2). Jeżeli termin *obiekt danych* nie jest jeszcze zbyt jasny, postaraj się wstawić poprzez schowek fragment jakiegoś arkusza kalkulacyjnego (może być to tabela), pochodzącego np. z Excela, do pliku dokumentu edytora tekstu, np. Worda. Właśnie wykonałeś operację wymiany obiektu danych pomiędzy dwiema niezależnie działającymi aplikacjami. Dane możemy wymieniać za pośrednictwem schowka (DDE, ang. *Dynamic Data Exchange*), czyli mechanizmu dynamicznej wymiany danych lub dużo bardziej wydajnego, jednolitego transferu danych UTD (ang. *Uniform Data Transfer*), lub też na zasadzie zwykłego przeciągania. W żargonie informatycznym tę ostatnio wymienioną operację określono by mianem *drag and drop*. W dosłownym tłumaczeniu brzmi to trochę zabawnie: zawlec (ang. *drag*) i zrzucić (ang. *drop*).

OLE Automation

Jest częścią standardu OLE 2. Umożliwia zapisywanie w aplikacji sekwencji działań OLE w postaci ciągu poleceń, które dany program ma zinterpretować.

Model COM

Component Object Model jest standardem pozwalającym współdzielić obiekty z wieloma aplikacjami. Określa też zasady komunikacji pomiędzy obiektami. Obiekty takie muszą być rozróżnialne już na poziomie systemu operacyjnego. Z reguły reprezentowane są w postaci plików wykonawczych z rozszerzeniem *.exe* lub bibliotek z rozszerzeniem *.dll*. Pewnym uogólnieniem COM jest technologia DCOM (ang. *Distributed COM*) pozwalająca wykorzystywać obiekty fizycznie znajdujące się na innych komputerach połączonych w sieć.

Technologia ActiveX

Umożliwia współdzielenie obiektów z wieloma aplikacjami, jak również umieszczanie obiektów w sieci Internet.

Środowisko programisty — IDE

Zintegrowane środowisko programisty stanowi zbiór wszystkich niezbędnych narzędzi pomocnych w błyskawicznym projektowaniu i uruchamianiu aplikacji. Zasadnicze elementy, które wchodzą w skład IDE to:

- Główne menu.
- Pasek narzędzi.
- Główny formularz.
- Okno edycji kodu.
- Inspektor obiektów (ang. *Object Inspector*).

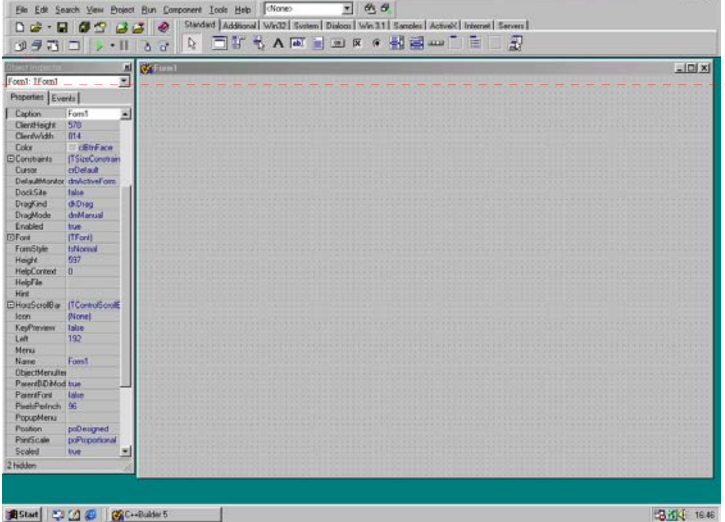
Odrębną grupę narzędzi pomocnych w szybkim tworzeniu aplikacji stanowią komponenty VCL. Jednak ze względu na swoje znaczenia zostaną one omówione w osobnym rozdziale.

Usunięto: ,
Usunięto: b
Usunięto: a
Komentarz: Może raczej umiejscawianie lub umieszczanie.
Komentarz: Oznaczyłam kursywą.
Usunięto: dla nas
Usunięto: P
Usunięto: W
Usunięto: ,
Usunięto: (
Komentarz: Zdaje się, że przyjęło się tłumaczyć te słowa jako „przeciągnij i upuść" i z tego co wiem, też jest to dosłowne tłumaczenie.
Usunięto: pomiędzy
Usunięto: e
Usunięto: i.
Usunięto: przez
Usunięto: e
Usunięto: i
Usunięto: ich
Komentarz: Czy to poprawne sformułowanie?
Komentarz: Co to jest IDE?
Usunięto: –
Komentarz: Czy to poprawne sformułowanie?
Usunięto: W skład IDE wchodzą następujące główne elementy:
Usunięto: paleta
Usunięto: ów
Usunięto: zdecydujemy się poświęcić jej osobny rozdział.

Po uruchomieniu programu C++Builder 5 okno monitora powininno wyglądać podobnie jak na rysunku 1.1. Może zdarzyć się i taka sytuacja, że formularz o nazwie Form1 nie pojawi się od razu, wówczas należy z głównego menu wybrać opcję File|New Application.

Usunięto: en

Rys. 1.1. *Zintegrowane środowisko programisty IDE C++ Buildera 5*

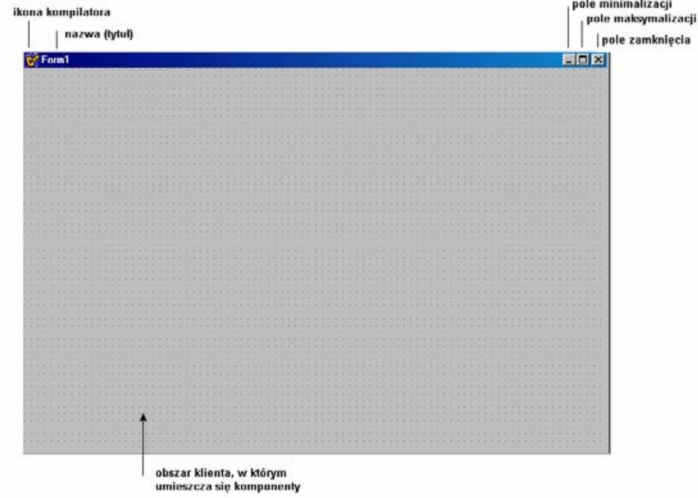


Komentarz: Czy to poprawne określenie?

Usunięto: –

Centralną część monitora zajmować będzie obszar zwany formularzem (w bardziej zaawansowanych opracowaniach obszar ten określa się mianem obszaru klienta), którego nazwa domyślnie przyjmowana jest jako Form1 (formularz, forma 1). Formularz posiada wszystkie cechy standardowego okna Windows. Już w tym momencie możemy uruchomić naszą aplikację naciskając chociażby przycisk F9. Na pewno zauważymy, że po uruchomieniu tego programu zachowuje się on tak samo jak każda aplikacja Windows.

Rys. 1.2. *Elementy standardowego formularza C++ Buildera*



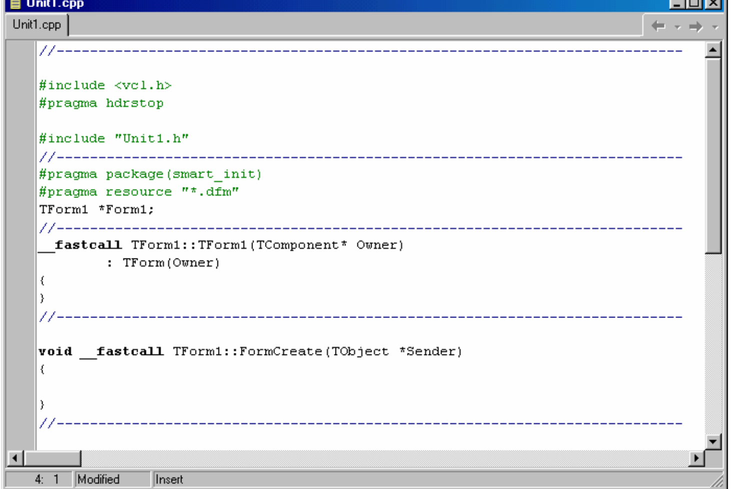
Jeżeli ktoś dokonał swojego pierwszego historycznego uruchomienia aplikacji, niech jak najszybciej ją zamknie klikając oczywiście w pole zamknięcia. Już niedługo nauczymy się umieszczać na formularzu różne komponenty, ale tymczasem wykażmy się odrobiną cierpliwości. Aby dostać się do kodu głównego modułu formularza wystarczy go kliknąć dwa razy. Ujrzymy wówczas okno edycji kodu podobne do pokazanego na rysunku 1.3.

Usunięto: tym czasem

Usunięto: dwa razy

Usunięto: na nim

Usunięto: .



Rys. 1.3. *Okno edycji kodu*

Być może powyższe zapisy jeszcze niewiele nam mówią, ale stopniowo będziemy je rozszyfrowywać. Właśnie tutaj będziemy pisać teksty naszych programów. Należy jednak pamiętać, że każda nauka programowania w Windows musi rozpocząć się od poznawania środowiska, w którym przyjdzie nam pracować, w naszym wypadku — C++Buildera 5.

Usunięto: –

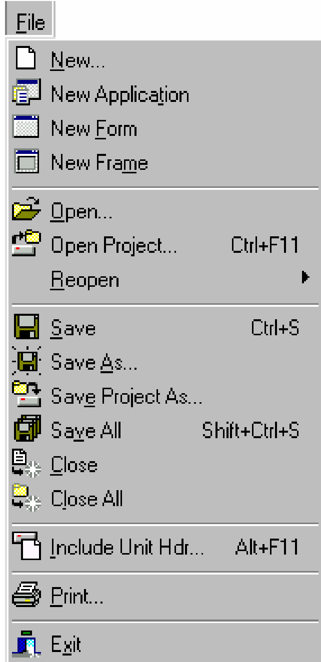
Struktura głównego menu

Rys. 1.4. *Główne menu*

Menu File

Korzystając z Menu File mamy do dyspozycji następujące opcje:

Rys. 1.5. Menu File



New...
Polecenie tworzy nowy projekt, formularz, okno dialogowe lub otwiera przykładowe projekty aplikacji opcja File|New|Projects.

Usunięto: –

New Application
Polecenie utworzenia nowego projektu. Nowo powstały projekt składa się z pustego formularza o nazwie Form1 oraz odpowiadającego mu modułu o nazwie *Unit1.cpp*.

New Form
Polecenie utworzenia nowego, pustego formularza.

Usunięto: U

Usunięto: e

New Frame
Polecenie utworzenia nowej ramki.

Open...
Polecenie otwarcia modułu, obiektu lub projektu. Katalogiem domyślnym będzie katalog, w którym zainstalowany jest Builder.

Open Project...
Polecenie otwarcia zapisanego wcześniej na dysku projektu.

Reopen
Wyświetlenie listy ostatnio używanych projektów, z których każdy można natychmiast otworzyć.

Usunięto: Zostaje wyświetlona

Usunięto: a

Save
Polecenie zapisania bieżącego modułu na dysku. Domyślnie plik ten będzie miał rozszerzenie **.cpp*.

Save As...
Zapisanie wybranego modułu pod nową nazwą. Zawsze dobrym zwyczajem jest zapisywanie kolejnych modułów pod innymi nazwami.

Usunięto: Zapisuje

Usunięto: y

Save Project As...
Polecenie zapisania aktualnie używanego projektu pod inną nazwą.

Save All
Zapisanie na dysku wszystkich aktualnie otwartych plików C++Buildera.

Close
Zamknięcie aktualnie używanego modułu kodu **.cpp* wraz z odpowiadającym mu formularzem.

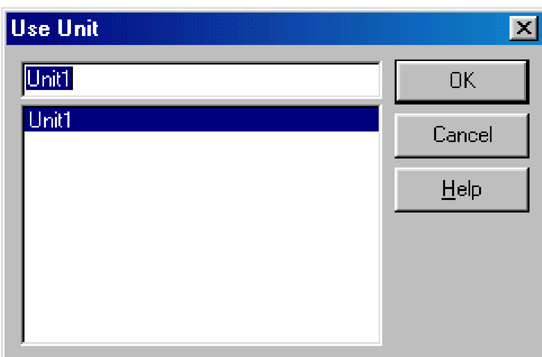
Close All
Zamknięcie aktualnie otwartego projektu.

Usunięto: yka

Usunięto: y

Include Unit Hdr...
Dołączenie do aktualnie używanego modułu kodu nowego pliku nagłówkowego. Jeżeli aktualnie pracujemy z formularzem Form2, któremu odpowiada moduł *Unit2.cpp* i zechcemy dołączyć moduł powiedzmy *Unit1.cpp*, wówczas użycie tego polecenia spowoduje wyświetlenie następującego okna:

Usunięto: informacji o następującej treści:



Rys. 1.6. Dołączanie nowego modułu

Określimy w ten sposób, czy moduł *Unit1.cpp* ma być używany przez *Unit2.cpp*.

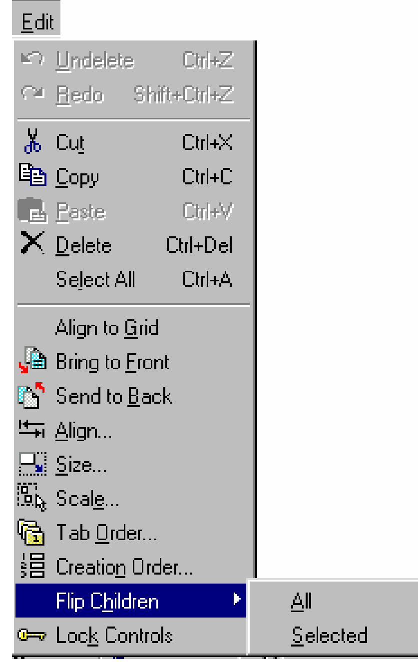
Print...
Polecenie drukowania aktualnie używanego elementu projektu. Gdy zechcemy wydrukować zawartość okna edycji kodu pojawi się opcja Print Selection. W przypadku drukowania formularza ujrzymy okienko Print Form.

Exit
Opuszczenie C++Buildera i ewentualne zapisanie wszystkich otwartych elementów aplikacji.

Menu Edit

Pełne rozwinięcie menu edycyjnego pokazano na rysunku 1.7.

Rys. 1.7. Menu Edit



Undelete

Podobnie jak we wszystkich standardowych aplikacjach Windows, opcja ta pozwala na anulowanie ostatniej operacji. Jeżeli przez pomyłkę usunięto jakiś komponent z formularza, używając Undelete możemy cofnąć usuwanie.

Redo

Polecenie odwrotne w stosunku do Undelete.

Cut

Umieszczanie zaznaczonego komponentu lub tekstu w schowku.

Usunięto: y

Copy

Polecenie kopiowania zaznaczonego elementu do schowka. W schowku zostanie umieszczona jedynie jego kopia.

Paste

Wstawianie uprzednio skopiowanego do schowka obiektu (tekstu, komponentu) we wskazane miejsce pola edycji kodu lub formularza.

Usunięto: y

Delete

Usuwanie zaznaczonego obiektu. Operacja odwrotna możliwa jest przy użyciu Undelete.

Usunięto: Z

Usunięto: y

Usunięto: zostanie usunięty.

Select All

W przypadku edycji kodu źródłowego — zaznaczenie całego tekstu. W przypadku formularza — zaznaczenie wszystkich znajdujących się tam komponentów.

Usunięto: a

Usunięto: y

Usunięto: a

Usunięto: e

Usunięto: e

Usunięto: y

Align to Grid

Przy pomocy tego polecenia dopasowujemy położenia wszystkich elementów składowych formularza do jego siatki. Operacja ta będzie dawać widoczne efekty pod warunkiem odznaczenia opcji Snap to Grid w menu Tools|Environment Options|Preferences.

Usunięto: ,

Bring to Front

Zaznaczony element nie będzie ewentualnie przykrywany przez inne, znajdujące się w formularzu. Element taki będzie zawsze całkowicie widoczny.

Usunięto: komponent
Usunięto: na
Usunięto: Komponent

Send to Back

Polecenie odwrotne do Bring to Front.

Align.. Wywołanie polecenia w stosunku do uprzednio zaznaczonego komponentu umożliwia dopasowanie i wyrównanie jego położenia na formularzu.

Usunięto: ¶
Komentarz: Czy chodzi o „Dopasowanie i wyrównanie fragmentu zaznaczonego wcześniejszym poleceniem.”?

Size...

Dokładne ustalenie rozmiaru obiektu. Operacja ta może być z powodzeniem użyta w stosunku do uprzednio zaznaczonej grupy obiektów.

Usunięto: Umożliwia
Usunięto: d
Usunięto: komponentu.
Usunięto: komponentów.

Scale...

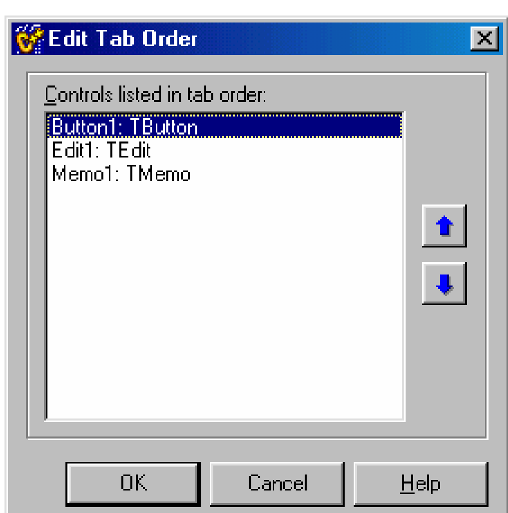
Polecenie przeskalowania formularza jako całości wraz ze wszystkimi elementami wchodzącymi w jego skład.

Usunięto: komponentami

Tab Order...

Pisząc aplikacje do Windows w wielu wypadkach staramy się uniezależnić od działania myszki. Istnieje możliwość ustalenia kolejności przechodzenia pomiędzy składnikami formularza przy użyciu klawisza Tab. Polecenie Tab Order wyświetla okienko dialogowe pokazane na rys. 1.8. Używając przycisków opatrzonych strzałkami można w prosty sposób ustalić kolejność przechodzenia pomiędzy wszystkimi aktualnie dostępnymi elementami, które wchodzą w skład projektowanego formularza.

Usunięto: komponentami
Usunięto: komponentami



Rys. 1.8. Okno dialogowe Edit Tab Order

Creation Order...

Opcja pozwalająca ustalić kolejność tworzenia tzw. komponentów niewidocznych (przestają być widoczne w momencie uruchomienia aplikacji).

Flip Children

Umożliwienie automatycznej zamiany kolejności ułożenia poszczególnych części formularza.

Usunięto: a
Usunięto: ą
Usunięto: ę
Usunięto: komponentów
Usunięto: na
Usunięto: u.

Lock Controls

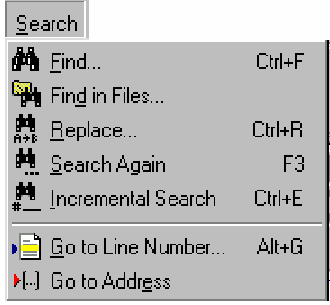
Wybierając tę opcję zablokujemy możliwość przemieszczania obiektów w obrębie formularza tworzonej aplikacji. Wybranie Lock Controls zapobiega przypadkowej zmianie położenia już wybranego obiektu.

Usunięto: ą
Usunięto: komponentów
Usunięto: komponentu.

Menu Search

Pokazane w rozwinięciu na rys. 1.9 menu Search zawiera następujące opcje:

Rys. 1.9. Menu Search



Find...

Wyszukanie w kodzie wybranego fragmentu tekstu. Przy pomocy okna dialogowego Find Text określamy żądane parametry wyszukiwania.

Usunięto: Opcja pozwalająca wyszukać
Usunięto: y

Find in Files...

Opcja ta umożliwia przeszukiwanie plików. Przy pomocy zakładki Find in Files określamy żądane parametry wyszukiwania.

Usunięto: U

Replace...

Wyszukanie określonego tekstu lub jego fragmentu i zastąpienie go innym.

Search Again

Wyszukanie kolejnego wystąpienia określonego tekstu lub jego fragmentu.

Incremental Search

Jest to tzw. opcja niewidoczna. Przed skorzystaniem z jej usług najlepiej jest ustawić kursor na samym początku tekstu kodu. Po wybraniu Search|Incremental Search należy zacząć pisać szukane słowo. Builder odczyta pierwszą literę i natychmiast przeniesie kursor do pierwszego napotkanego w tekście zwrotu zawierającego wpisaną literę.

Go to Line Number...

Przeniesienie kursora do wskazanego wiersza kodu.

Go to Address

Opcja dostępna w trakcie działania aplikacji. Umożliwia krokowe sprawdzanie wartości zmiennych, rejestrów CPU itp. Po pojawieniu się okienka dialogowego, podobnego do pokazanego na rys. 1.10, należy wpisać żądaną wartość. Liczby heksadecymalne należy poprzedzić parą znaków `0x`.

Usunięto: 20

Rys. 1.10. Okno dialogowe Enter Address to Position to

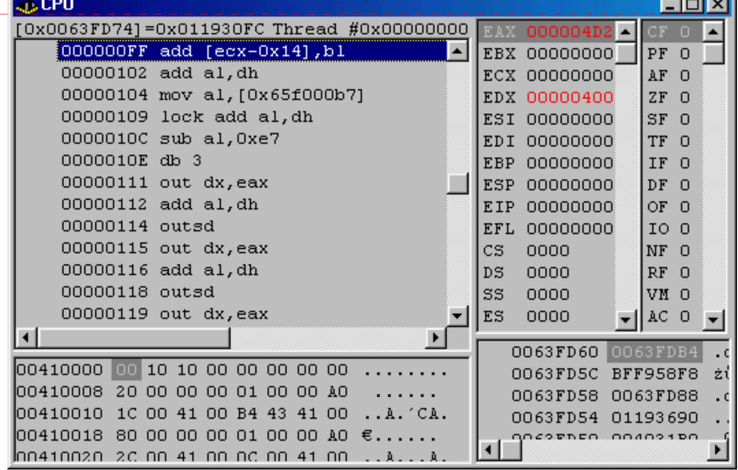
Usunięto: 2

Potwierdzając przyciskiem OK. zobaczymy okno aktualnego stanu m.in. rejestrów CPU (ang. *Central Processing Unit*) czyli jednostki centralnej lub po prostu procesora. Poruszanie się w oknie CPU możliwe jest dzięki kombinacji klawiszy Ctrl+(prawa/lewa) strzałka.

Usunięto: m. in.

Rys. 1.11. Okno CPU

Usunięto: 2



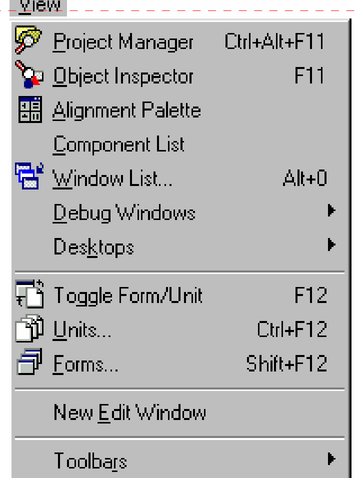
Menu View

Przedstawione na rysunku 1.12 menu View zawiera następujące opcje:

Usunięto: 22

Rys. 1.12. Menu View

Usunięto: 2



Project Manager

Polecenie to wywołuje menedżera projektów.

Usunięto: W

Object Inspector

To polecenie wywołuje inspektora obiektów.

Usunięto: P

Alignment Palette

Opcja umożliwiająca wzajemne ułożenie i dopasowanie komponentów na formularzu. Jest to graficzny odpowiednik opcji Edit|Align.

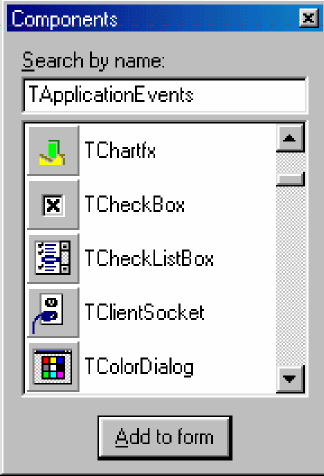
Component List

Użycie tego polecenia powoduje uaktywnienie okna (rys. 1.13) zawierającego wszystkie aktualnie dostępne komponenty. Są one ułożone w porządku alfabetycznym. Za pomocą przycisku Add to form dowolny komponent można dodać do formularza.

Usunięto: 23

Rys. 1.13. Lista komponentów

Usunięto: 2



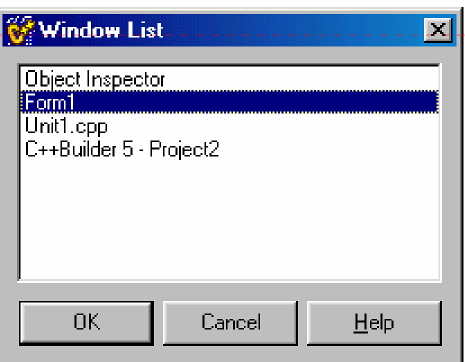
Window List...

Użycie tego polecenia powoduje uaktywnienie dialogu, w którym pokazana jest lista aktualnie otwartych okien (rys. 1.14). Zaznaczając odpowiednią pozycję można przenieść się do wybranego okna.

Usunięto: 24

Rys. 1.14. Lista aktualnie otwartych okien

Usunięto: 2



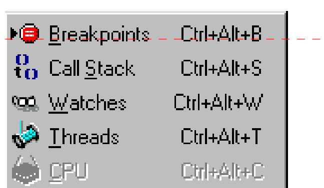
Debug Windows

W skład Debug Windows wchodzi lista poleceń pokazana na rysunku 1.15 .

Usunięto: 25.

Rys. 1.15. Opcje Debug Windows

Usunięto: 2



- Breakpoints — wyświetla listę pułapek pomocnych w śledzeniu programu korzystając z debuggera, czyli programu uruchomieniowego. Przy pomocy tego programu mamy

Usunięto: –

Usunięto: (ang. *breakpoint*)

Komentarz: Lista pułapek pomocnych w śledzeniu programu? Czy debuggera używa się do wyświetlenia pułapek, czy do śledzenia programu?

Usunięto: którego

możliwość pracy krok po kroku oraz możliwość sprawdzania wartości zmiennych i rejestrów procesora.

- Call Stack — opcja ułatwiająca ustalenie kolejności wywoływania funkcji głównego programu podczas działania programu uruchomieniowego.
- Watches — wyświetla okno Watch List, w którym można oglądać aktualne wartości wyrażeń lub zmiennych. Stosowana jest podczas operacji śledzenia wykonywania programu.
- Threads — W okienku Thread Status pojawi się lista aktualnie uruchomionych wątków.
- CPU — wyświetla okienko aktualnego stanu CPU. Opcja ta jest aktywna w czasie działania programu.

Desktops
Użycie tego plecenia umożliwia skonfigurowanie i zapisanie pod wybraną nazwą wymaganego przez użytkownika wyglądu pulpitu (ang. *desktop*). Opcje tego podmenu pokazane są na rysunku 1.16.

Rys. 1.16. Opcje Desktops

Toggle Form/Unit
Możliwość przełączenia (ang. *toggle*) pomiędzy edycją formularza a odpowiadającym mu oknem edycji kodu (por. rysunki 1.1 oraz 1.3).

Units...
Polecenie to podaje listę wszystkich modułów należących do projektu.

Forms...
Ogólnie rzecz biorąc, w skład aplikacji może wchodzić wiele formularzy. Przy pomocy tego polecenia można wyświetlić listę wszystkich formularzy używanych przez aplikację.

New Edit Window
Polecenie otwarcia kolejnego okna edycji kodu. Dzięki temu możemy pracować z dwoma modułami jednocześnie.

Toolbars
Możliwość konfiguracji struktury głównego menu. Jeżeli wszystkie opcje Toolbars będą zaznaczone (rys. 1.17), to główne menu będzie wyglądać tak jak na rysunku 1.4.

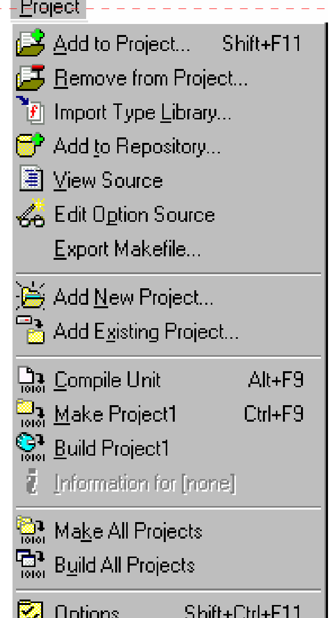
Rys. 1.17. Opcje Toolbars

Menu Project

W skład tego menu wchodzą następujące, pokazane na rys. 1.18, opcje:

Usunięto: krokowej
Usunięto: e
Usunięto: –
Usunięto: –
Usunięto: –
Usunięto: –
Usunięto: 26.
Usunięto: 2
Usunięto: P
Usunięto: Polecenie to wyświetla
Usunięto: 27
Usunięto: 2
Usunięto: 28

Rys. 1.18. Opcje Menu Project



Usunięto: 2

Add to Project...
Opcja ta umożliwia włączenie wskazanego modułu do projektu modyfikując automatycznie plik z opisem projektu.

Komentarz: „i dostosowanie pliku do opisu projektu"?

Remove from Project...
Usuwa wybrany moduł z projektu modyfikując jednocześnie plik główny projektu.

Import Type Library...
Umożliwia zarejestrowanie w środowisku Buildera wybranej biblioteki, która od tej chwili będzie traktowana jak każda składowa biblioteki VCL.

Add to Repository...
Aktualnie wykorzystywany formularz będzie umieszczony w repozytorium.

View Source
Polecenie edycji kodu projektu.

Edit Option Source
Polecenie edycji wszystkich informacji dotyczących projektu oraz edycji przypisań i odwołań do plików i bibliotek z nim związanych. Będą wyświetlane m.in. informacje o środowisku, kompilatorze, standardzie kodu, nazwie pliku wynikowego itp.

Usunięto: związanych z
Usunięto: em
Usunięto: z nim.
Usunięto: m. in.
Usunięto: ,

Export Makefile...
Zapisanie pliku do kompilacji projektu (tzw. pliki *makefile*). Plik taki składa się z ciągu znaków ASCII i zawiera zestaw instrukcji do kompilacji projektu.

Add New Project...
Polecenie tworzy nowy projekt w grupie projektów. Opcja ta działa podobnie jak View|Project Manager|New.

Add Existing Project...
Przy pomocy tego polecenia można dodać do grupy projektów projekt już istniejący i zapisany wcześniej na dysku.

Usunięto: Dodaje

Compile Unit
Kompilacja modułu projektu.

Make Project1
Kompilacja aktualnego projektu w tzw. trybie *Make*. Kompilator kompiluje kody źródłowe wszystkich modułów wchodzących w skład projektu, w których dokonano zmian od czasu ostatniej kompilacji. Na dysku w aktualnym katalogu zostanie utworzony program wykonywalny.

Komentarz: Czy chodzi o: „To polecenie spowoduje połączenie kodów źródłowych wszystkich tych modułów projektu, w których dokonano zmian od czasu ostatniej kompilacji."

Komentarz: Czy to prawidłowe określenie?

Build Project1
Polecenie kompilacji aktualnego projektu w tzw. trybie *Build*. Kompilowane będą wszystkie moduły niezależnie od tego czy były ostatnio modyfikowane, czy nie. Na dysku w aktualnym katalogu zostanie utworzony plik wykonywalny.

Information for (...)
Podaje informacje na temat ostatnio skompilowanego projektu, liczba linii, rozmiar w bajtach: danych, rozmiar kodu, rozmiar pliku wykonywalnego, itp.

Komentarz: Czy chodzi o: „Podanie informacji na temat ostatnio skompilowanego projektu (chodzi o liczbę linii, rozmiar kodu, rozmiar pliku wykonywalnego itp.)

Make All Projects
Kompilacja w trybie Make wszystkich projektów wchodzących w skład grupy projektów.

Build All Projects
Kompilacja w trybie Build wszystkich projektów wchodzących w skład grupy projektów.

Options...
Polecenie wywołania okna dialogowego Project Options, w którym można ustalić parametry kompilatora i konsolidatora.

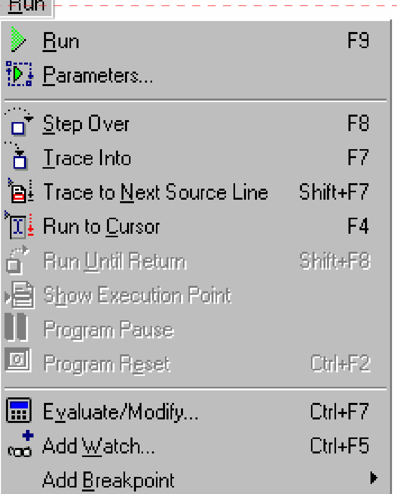
Menu Run

Wymienione menu zawiera opcje pokazane na rysunku 1.19.

Usunięto: 28.

Rys. 1.19. Opcje Menu Run

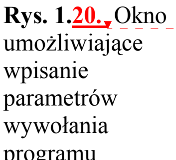
Usunięto: 28.



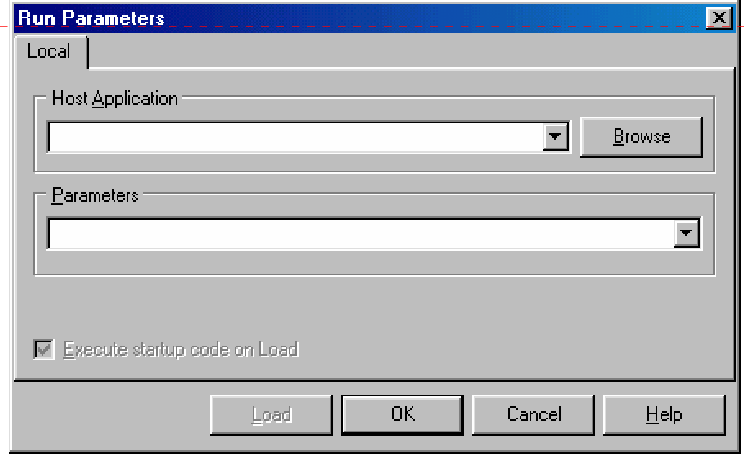
Run
Polecenie dokonania kompilacji (jeżeli jest to wymagane) z jednoczesnym uruchomieniem aplikacji.

Parameters...
Polecenie to wyświetla okno dialogowe (rys. 1.20), w którym można ustalić parametry wywołania aplikacji.

Usunięto: 29



Rys. 1.20. Okno umożliwiające wpisanie parametrów wywołania programu



Usunięto: 29.

Step Over
Uruchomienie aplikacji w trybie krokowym z możliwością śledzenia jej przebiegu wiersz po wierszu. Wywołania funkcji traktowane będą jako jedna instrukcja bez zaglądania do ich wnętrza.

Komentarz: ?

Trace Into
Uruchomienie aplikacji w trybie krokowym. W momencie wywołania funkcji przenosimy się do jej wnętrza.

Usunięto: napotkania

Komentarz: ?

Trace to Next Source Line
Uzupełnienie poprzedniej opcji o możliwość zobaczenia kolejnego wiersza kodu, który jest wykonywany.

Run to Cursor
Polecenie wykonania programu do miejsca, w którym ustawiliśmy kursor. Wartość zmiennej można zobaczyć używając polecenia View|Debug Windows|Watches.

Run Until Return
Krokowe śledzenie wykonywania programu do momentu uruchomienia aplikacji.

Show Execution Point
Jeżeli w czasie uruchomienia aplikacji w trybie krokowym okno edycji kodu zostało zamknięte, przy pomocy tego polecenia okno zostanie otwarte, zaś kursor znajdować się będzie w wierszu, który jest aktualnie wykonywany.

Program Pause
Tymczasowe wstrzymanie uruchomionego programu.

Program Reset
Polecenie zatrzymania wykonywanego programu z jednoczesnym usunięciem go z pamięci.

Evaluate/Modify...
W czasie działania debuggera istnieje możliwość nie tylko oglądania zmiennych i parametrów, ale również modyfikowania ich wartości. Można też obliczać wyrażenia zawierające te zmienne lub parametry.

Usunięto: można też

Usunięto: ć

Add Watch...
Dodanie nowej zmiennej lub parametru do listy Watches.

Add Breakpoint
Założenie pułapki. Wskazany wiersz kodu zostanie podświetlony.

Menu Component

Pokazane na rysunku 1.21 menu posiada następujące opcje:

Rys. 1.21. Menu Component

Component
New Component...
Install Component...
Import ActiveX Control...
Create Component Template...
Install Packages...
Configure Palette...

New Component...
Wywołanie zakładki New Component, pomocnej w utworzeniu własnego komponentu.

Install Component...
Polecenie to dodaje nowy komponent do biblioteki VCL.

Import ActiveX Control...
Polecenie dołączenia zarejestrowanego oraz istniejącego obiektu ActiveX do wybranego pakietu VCL.

Create Component Template...
To polecenie tworzy szablon komponentów. Kilka elementów można połączyć i korzystać z nich tak, jakby były pojedynczym obiektem.

Install Packages...
Opcja umożliwiająca odpowiednie zarządzanie pakietami (ang. *packages*), które stanowią część środowiska i z których zbudowana jest biblioteka VCL. Pakiety takie można dodawać, usuwać, edytować, tak jak pokazuje to rys. 1.22.

Usunięto: 30

Usunięto: 30.

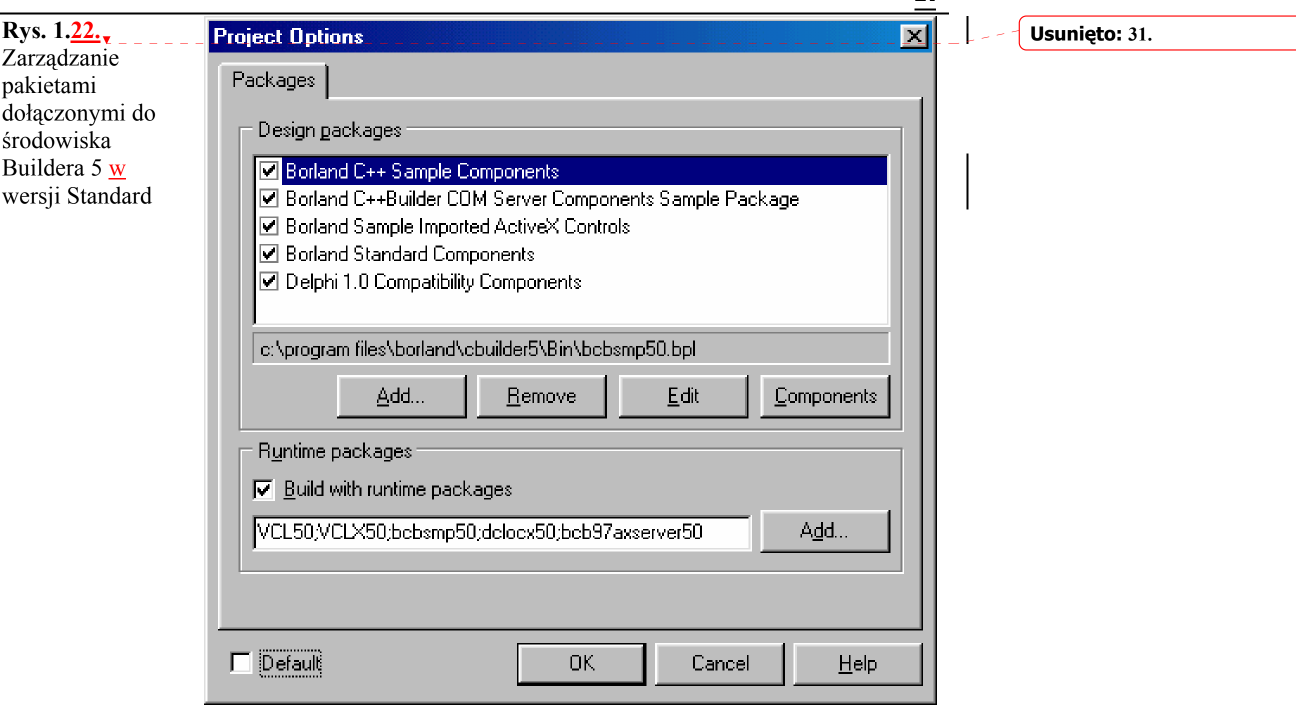
Komentarz: Czy może lepiej „Wywołanie zakładki New Component powoduje utworzenie nowego elementu formularza".

Usunięto: do wybranego pakietu VCL

Usunięto: komponentów

Usunięto: poddawać edycji ich zawartości

Usunięto: 31



Rys. 1.22. Zarządzanie pakietami dołączonymi do środowiska Buildera 5 w wersji Standard

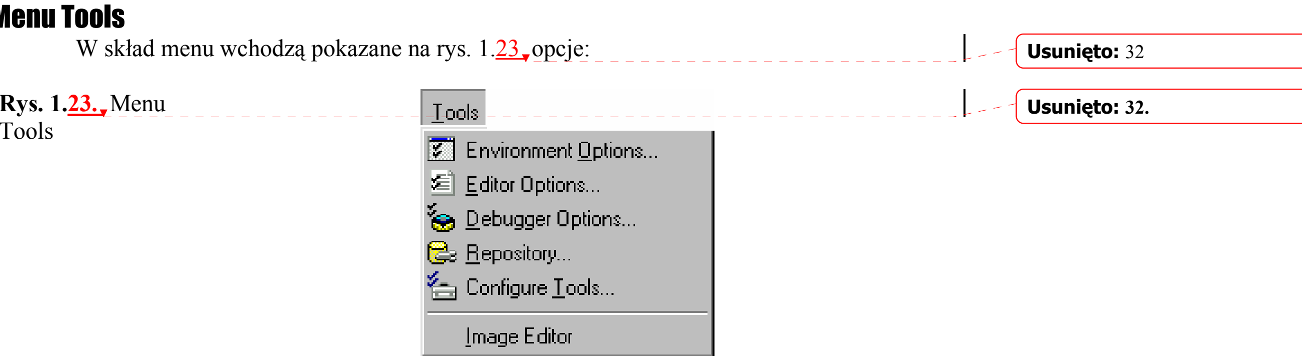
Usunięto: 31.

Configure Palette...
Możliwość dowolnego skonfigurowania układu palety komponentów poprzez ich dodawanie, usuwanie czy umieszczanie w innych miejscach.

Menu Tools

W skład menu wchodzą pokazane na rys. 1.23 opcje:

Usunięto: 32



Rys. 1.23. Menu Tools

Usunięto: 32.

Environment Options...
Opcja pomocna w określeniu parametrów konfiguracyjnych środowiska.

Editor Options...
Opcja umożliwiająca określenie w oknie edycji wielu parametrów konfiguracyjnych, takich jak: rodzaj czcionki, jej kolor, rozmiar okna itp.

Usunięto: okna edycji,

Debugger Options...
Ustalenie opcji debuggera.

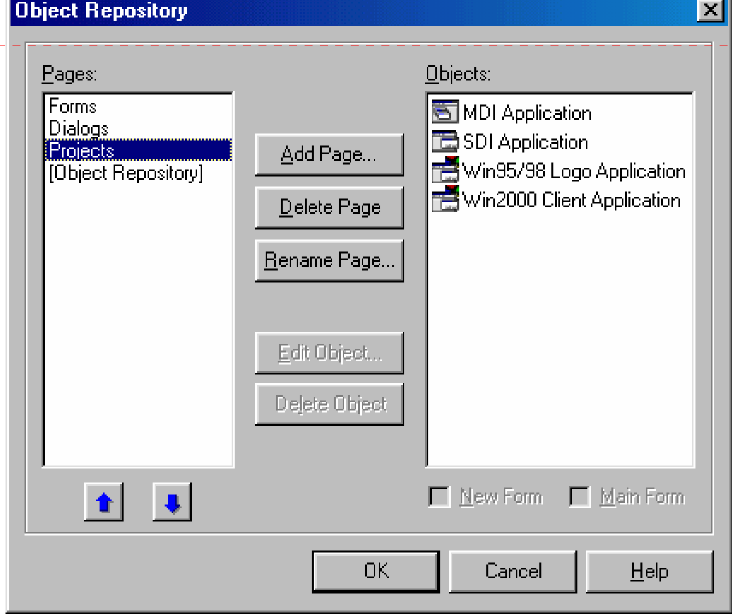
Repository...

Repozytorium jest centralnym systemem informacji o obiektach tworzących aktualny projekt. Dzięki tej opcji (rys. 1. 24.) można obiekty takie edytować, dodawać i usuwać.

Usunięto: 33

Usunięto: ,

Usunięto: 33.



Rys. 1.24. Repozytorium obiektów

Configure Tools...

To polecenie umożliwia odpowiednie skonfigurowanie środowiska.

Usunięto: U

Image Editor

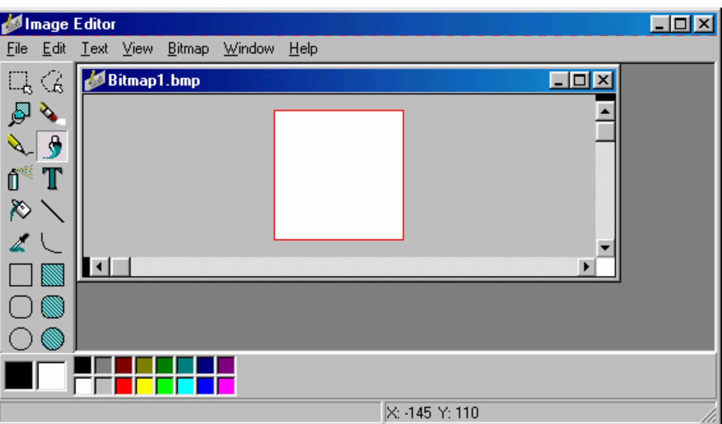
Edytor graficzny służy do samodzielnego projektowania ikon, przycisków, różnego rodzaju rysunków pomocnych w projektowaniu aplikacji. Na rys. 1. 25 pokazano wygląd edytora. Zasada jego obsługi w niczym nie odbiega od zasady posługiwania się takimi aplikacjami, jak Paint czy Paint Brush.

Usunięto: ący

Usunięto: 34

Usunięto: chociażby Paintem czy Paint Brushem.

Usunięto: 34.



Rys. 1.25. Edytor graficzny C++Buildera w działaniu

Menu Help

Przedstawione w rozwinięciu na rys. 1.26 menu posiada następujące opcje:

Usunięto: 35

Rys. 1.26. Menu Help

Usunięto: 35.

C++Builder Help

C++Builder Tools

Windows SDK
Zawierają spisy treści oraz pliki pomocy C++ Buildera 5 i Win32 API.

Komentarz: Czy to zdanie odnosi się tylko do polecenia „Windows SDK", czy też do wszystkich trzech wymienionych poleceń?

Borland Home Page

Borland Community Page

C++Builder Home Page

C++Builder Developer Support

C++Builder Direct...
Polecenia te pozwalają na automatyczne połączenie ze stronami WWW firmy Borland oraz stronami poświęconymi C++Builderowi 5.

Komentarz: Które polecenia?

Usunięto: i

About...
Przytrzymując lewy klawisz Alt Napisz: DEVELOPERS.

Komentarz: Czy nie jest to zbyt skąpa charakterystyka polecenia „About..."?

Menu Desktop

Przy pomocy zestawu opcji widocznych na rysunku 1. 27 możemy zapisać samodzielnie skonfigurowany pulpit środowiska C++Builder 5.

Usunięto: 36

Rys. 1.27. Menu Desktop

Usunięto: 36.

Pick List
Zawiera listę nazw, pod którymi zapisano wygląd skonfigurowanych pulpitów.

Save current desktop
Przy pomocy tego okienka dialogowego zapisujemy aktualnie skonfigurowany pulpit. Analogiczną operacją będzie View|Desktops|Save Desktop.

Set debug desktop
Przy pomocy tego polecenia można określić wygląd pulpitu podczas uruchamiania aplikacji np. poleceniem Run|Run (F9). Analogiczną operacją będzie View|Desktops|Set Debug Desktop. Wszystkie dane o dokonanej konfiguracji pulpitu zostaną zapisane na dysku w pliku z rozszerzeniem *.dst*.

Usunięto: Zapisuje

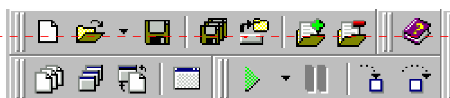
Pasek narzędzi — Speed Bar

Pokazany na rysunku 1.28 pasek narzędzi pozwala na szybszy dostęp do najczęściej używanych poleceń IDE Buildera. Standardowo zawiera on 16 przycisków, które są najczęściej używane przez programistów. Przyciski te pogrupowane są w czterech obszarach (por. rys. 1.27):

- Standard
- View
- Debug
- Custom

Oczywiście, dostęp do każdego z nich możliwy jest również z poziomu głównego menu.

Rys. 1.28. Pasek narzędzi



Inspektor obiektów — Object Inspector

Inspektor obiektów jest bardzo ważną częścią IDE. Posługując się nim możemy bardzo szybko ustalać i zmieniać cechy obiektów. Możemy też w wygodny sposób zarządzać i edytować metody stanowiące odpowiedź na określone zdarzenie. Zasadniczą częścią inspektora obiektów są dwie zakładki, czyli karty: karta właściwości, cech (ang. *properties*) oraz karta obsługi zdarzeń (ang. *events*).

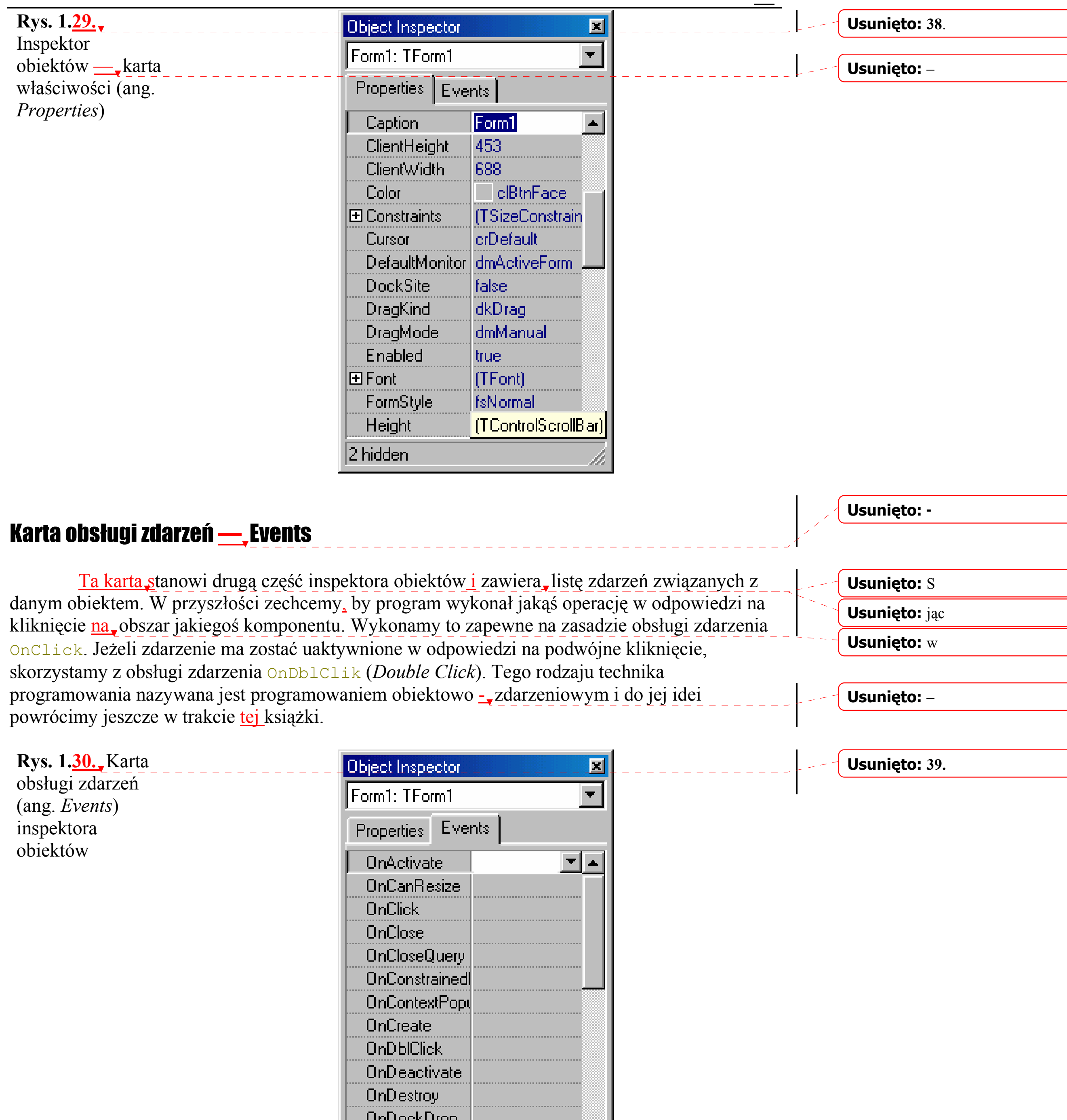
Karta właściwości — Properties

Karta właściwości pokazana jest na rysunku 1.38. Umożliwia ona wygodne edytowanie właściwości samego formularza oraz aktualnie zaznaczonego na nim obiektu. Raz klikając na obszarze formularza wywołamy inspektora obiektów. Jeżeli teraz zechcemy zmienić nazwę formularza, wystarczy jego cesze `Caption` przypisać własną nazwę. Podobnie korzystając z cechy `Icon` możemy w prosty sposób zmienić ikonę formularza. Własną, oryginalną ikonę możemy stworzyć przy pomocy edytora graficznego pokazanego na rys. 1.25.

Niektóre właściwości poprzedzone są znaczkiem **+**. Oznacza to, że zawierają szereg zagnieżdżonych opcji. Dla przykładu rozpatrzmy cechę `BorderIcons`. Klikając na **+** zobaczymy kilka pozycji. Przypiszmy cesze `biMinimize` wartość `false`, a następnie poleceniem Run|Run lub F9 spróbujmy uruchomić aplikację. Pole minimalizacji stanie się wówczas nieaktywne. Podobnie cechom `biSystemMenu` oraz `biMaximize` możemy przypisać wartości `false`, jednak wówczas po uruchomieniu formularza będziemy mieli problem z jego zamknięciem (pole zamknięcia jest wygaszone — nieaktywne). W tego typu wypadkach należy użyć polecenia Run|Program Reset.

Możemy również już teraz ustalić np. kolor obszaru klienta — przy pomocy cechy `Color`, rozmiary formularza: wysokość i szerokość — przy pomocy cech `Height`, `Width`, a także położenie formularza na ekranie — przy pomocy cech `Top`, `Left`.

Usunięto: –
Usunięto: 37
Usunięto: 37.
Usunięto: –
Usunięto: Umożliwia
Usunięto: nie
Usunięto: cję
Komentarz: Edytować metody?
Usunięto: ych
Usunięto: (
Usunięto:)
Usunięto: –
Usunięto: P
Usunięto: znajdującego się na formularzu. Już teraz możemy zmienić wiele cech formularza pokazanych na rysunku 1.2.
Usunięto: w
Usunięto: ujrzymy w
Usunięto: ze
Usunięto: wszystkie jago cechy.
Usunięto: 34.
Komentarz: ?
Usunięto: , że składa się ona z kilu
Usunięto: uruchommy
Usunięto: –
Usunięto: ,
Usunięto: –
Usunięto: a
Usunięto: –
Usunięto: y
Usunięto: oraz jego
Usunięto: –
Usunięto: y



Rys. 1.29. Inspektor obiektów — karta właściwości (ang. *Properties*)

Karta obsługi zdarzeń — Events

Ta karta stanowi drugą część inspektora obiektów i zawiera listę zdarzeń związanych z danym obiektem. W przyszłości zechcemy, by program wykonał jakąś operację w odpowiedzi na kliknięcie na obszar jakiegoś komponentu. Wykonamy to zapewne na zasadzie obsługi zdarzenia `OnClick`. Jeżeli zdarzenie ma zostać uaktywnione w odpowiedzi na podwójne kliknięcie, skorzystamy z obsługi zdarzenia `OnDblClik` (*Double Click*). Tego rodzaju technika programowania nazywana jest programowaniem obiektowo -zdarzeniowym i do jej idei powrócimy jeszcze w trakcie tej książki.

Rys. 1.30. Karta obsługi zdarzeń (ang. *Events*) inspektora obiektów

Usunięto: 38.
Usunięto: –
Usunięto: -
Usunięto: S
Usunięto: jąc
Usunięto: w
Usunięto: –
Usunięto: 39.

Podsumowanie

W niniejszym rozdziale zapoznaliśmy się z częścią IDE, czyli środowiska programisty oferowanym nam przez Buildera 5. Dalsze jego elementy będziemy omawiać już przy okazji konkretnych przykładów wykorzystania komponentów z biblioteki VCL. Umiemy samodzielnie skonfigurować dla własnych potrzeb pulpit, oswoiliśmy się też z inspektorem obiektów oraz opcjami dostępnymi z poziomu głównego menu. Przed nami C++Builder 5.

Komentarz: Czy to poprawne sformułowanie?

Rozdział 2
Borland C++Builder 5. Pierwsze kroki

Skoro umiemy już, przynajmniej teoretycznie, korzystać z niektórych elementów środowiska Buildera, najwyższy czas, aby zapoznać się z językiem programowania, który stanowić będzie podstawę tworzonych przez nas w przyszłości aplikacji oraz z praktycznymi sposobami korzystania z IDE. Istnieje tylko jeden, skuteczny sposób, by tego dokonać — napisanie własnego programu.

Usunięto: –

Ogólna postać programu pisanego w C++

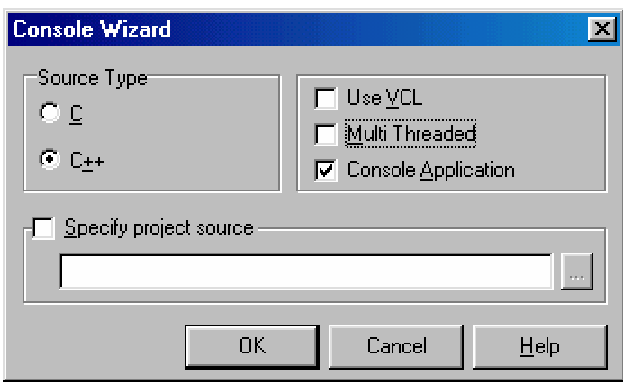
W niniejszym podrozdziale zapoznamy się z elementami składowymi programu pisanego dla Windows w języku C++. Wynikiem utworzenia takiego programu, inaczej mówiąc projektu, będzie plik wykonawczy *.exe* oraz kilka innych zbiorów danych bardzo pomocnych na etapie projektowania programu.
Wykonajmy na początek dwie proste czynności, mianowicie stwórzmy na dysku dwa oddzielne katalogi (foldery). Proponuję, by nazwać je po prostu `\Projekt01` oraz `\Projekt02`. W katalogach tych będziemy przechowywali pliki z których korzystać będą nasze dwie pierwsze aplikacje.

Sformatowane: Punktory i numeracja

Następnie uruchommy C++Buildera 5. Poleceniem File|New|Console Wizard otwórzmy nowy moduł. Inspektor obiektów powinien być nieaktywny, natomiast na pulpicie powinno pojawić się okno dialogowe podobne do tego z rysunku 2.1.

Usunięto: Następnie:¶ uruchommy C++Buildera 5,

Rys. 2.1. Console Wizard



W opcji Source Type zaznaczmy C++, zaś w drugim panelu odznaczmy Use VCL oraz wybierzmy Console Application. Zaznaczenie tej ostatniej opcji spowoduje, że nasz program będzie traktował główny formularz tak, jakby był normalnym okienkiem tekstowym DOS.

Potwierdzając przyciskiem OK od razu przejdziemy do okna (rys. 2.2), w którym będzie się znajdować szkielet kodu przyszłego programu.

Rys. 2.2. Kod modułu *Unit1.cpp*

```
Unit1.cpp
//---------------------------------------------------------------------------

#include <vcl.h>
#pragma hdrstop

//---------------------------------------------------------------------------

#pragma argsused
int main(int argc, char* argv[])
{
        return 0;
}
//---------------------------------------------------------------------------
1: 1   Modified   Insert
```

Chociaż powyższe zapisy być może dla niektórych z nas stanowić będą pewną niewiadomą, nie wnikajmy na razie w szczegóły, wszystko to dokładnie omówimy za chwilę. Tymczasem spróbujmy uzupełnić powyższy tekst, tak aby kompletny kod naszego modułu, nazwijmy go już jako *Unit01.cpp* wyglądał jak na wydruku 2.1. Następnie zapiszmy nasz moduł (polecenie File|Save As...) w katalogu *\Projekt01\Unit01.cpp*. Projekt modułu zapiszmy poleceniem File|Save Project As... w tym samym katalogu *\Projekt01\Projekt01.bpr*.

Wydruk 2.1. Kod modułu *Unit01.cpp* projektu *Projekt01.bpr*

```
#include <iostream.h>
#include <conio.h>
#pragma hdrstop

int main()
{
  cout << "Pierwszy program w C++";
  cout << endl << "Naciśnij klawisz...";
  getch();
  return 0;
}
//----------------------------------------------------------------------
```

Teraz spróbujmy uruchomić nasz program np. poleceniem Run|Run (F9). Nawet intuicyjnie poznamy, że po uruchomieniu, na ekranie w okienku udającym tryb tekstowy powinien pojawić cię napis: *Pierwszy program w C++*. Aby opuścić program wystarczy nacisnąć Enter.

Funkcja main()

Każdy program C lub C++ musi zawierać w sobie przynajmniej jedną funkcję. Główna funkcja `main()` jest tą, która zawsze musi istnieć w programie. Jest wywoływana jako pierwsza i powinna zawierać w sobie zestaw kolejnych instrukcji wykonywanych przez program, z reguły są to wywołania innych funkcji. Zestaw wszystkich instrukcji musi być zawarty pomiędzy parą nawiasów klamrowych `{ ... }`. Formalnie funkcja `main()` nie jest częścią C ani C++, jednak traktowana jest jako integralna część środowiska. W ogólnym wypadku C++ dopuszcza możliwość użycia parametrów formalnych w wywołaniu funkcji `main()`, w których mogą być zapisywane wartości ich argumentów, tak jak pokazuje to rysunek 2.2. Jednak ten sposób zapisu

głównej funkcji nie będzie nas interesował, również z tego powodu, że nigdy już do niego nie powrócimy w trakcie tej książki. Natomiast w większości spotykanych przypadków można postąpić w sposób dużo prostszy, zapisując `main()` w taki sposób, jak pokazano, to na wydruku 2.1. Jeżeli funkcja `main()` jest określonego typu (w naszym przypadku typu całkowitego `int`), to powinna zwrócić wartość tego samego typu. Tutaj wykonaliśmy tę operację poprzez instrukcję `return 0`, który to zapis jest niczym innym jak wartością powrotną udostępnianą w następstwie wywołania funkcji. Jeżeli funkcja byłaby typu `void` (tzw. typ pusty, pusta lista parametrów), to nie musi zwracać żadnej wartości.

Usunięto: y

Usunięto: ą

Instrukcja `return` zastosowana w funkcji `main()` zwraca do systemu operacyjnego kod zakończenia działania funkcji (programu). Wartość powrotna, udostępniana w następstwie wywołania funkcji, musi być liczbą całkowitą. W MS DOS oraz Windows 3x, 9x, NT, 2000 wartością tą jest `0` lub, co jest równoważne, wartość `FALSE`. Wszystkie pozostałe wartości będą sygnałem wystąpienia błędu. Podobną zasadą kierujemy się przy korzystaniu z różnych funkcji udostępnianych przez Win32 API.
Należy jednak pamiętać, iż bardzo wiele funkcji oferowanych w Win32 przez interfejs programisty jest typu `BOOL`, czyli mogących w wyniku wywołania zwrócić albo `TRUE` albo `FALSE`. Wówczas `TRUE`, czyli wartość niezerowa, określa prawidłowe zakończenie działania funkcji.
Podobną zasadę stosują niekiedy programiści przy określaniu wartości powrotnej funkcji, nie będącej częścią środowiska programistycznego lub systemu operacyjnego, czyli funkcji pisanej samodzielnie. Bardzo często jako kod powrotny wybieramy w takich wypadkach wartość `1` lub ogólnie `TRUE`.

Komentarz: Czy Win.3.2?

Komentarz: Interfejs programisty czy interfejs programowy?

Usunięto: ych

Należy pamiętać, że zarówno C, C++, jak i C++Builder, na ogół rozróżniają wielkość liter. Pewnym wyjątkiem są dane typu `TRUE` i `FALSE`. Tworząc aplikacje konsolowe przy pomocy C lub C++ należy je zapisywać małymi literami, czyli `true`, `false`. W C++Builderze jest to bez znaczenia.

Dyrektywa #include i prekompilacja

Pisząc w C lub C++ każdy program można zbudować posługując się jedynie prostymi instrukcjami oferowanymi przez te kompilatory. Należy jednak pamiętać, że zarówno C, jak i C++ nie posiadają wbudowanych instrukcji pozwalających na realizację operacji wejścia / wyjścia, czyli opcji umożliwiających wprowadzanie i wyprowadzanie na ekran, dysk lub inne urządzenie komunikatów użytkownika. Powoduje to konieczność wywoływania w odpowiednim miejscu programu różnych funkcji realizujących wymienione operacje wejścia / wyjścia. Większość takich funkcji znajduje się w plikach nagłówkowych C:

- *stdio.h* (ang. *standard library*) zawiera deklaracje typów i makrodefinicje wykorzystywane przez standardowe funkcje wejścia /wyjścia.
- *conio.h* (ang. *console input output*) zawiera deklaracje funkcji umożliwiających komunikację z konsolą. W przypadku programu przedstawionego na wydruku 2.1 funkcja `getch()` — reagująca na naciśnięcie klawisza, np. Enter — wymaga użycia *conio.h*.

Usunięto: jącej

Usunięto: jącej

Usunięto: ej

Wszystko to jest również aktualne w C++, niemniej jednak język ten może wykorzystywać słowo `cout` oraz operator `<<` (w omawianym kontekście znak graficzny `<<` nazywamy operatorem wyjścia lub wyprowadzania danych) pozwalające wyprowadzić (również na ekran) łańcuchy znaków oraz wartości, akceptowanych przez C++, typów danych. Sekwencja dowolnej liczby znaków ujętych w cudzysłów `"... "` nazywana jest ciągiem znaków, tekstem lub stałą tekstową. Instrukcja `endl` powoduje przesunięcie kursora do początku następnego wiersza. W tym wypadku wymagane jest użycie pliku nagłówkowego *iostream.h*. Bardzo często operator `cout` występuje w parze z operatorem `cin`, ten ostatni służy do wczytywania i zapamiętywania danych.

Zgodnie ze standardem ANSI każda funkcja biblioteczna musi być zadeklarowana w pewnym zbiorze nagłówkowym, którego zawartość włączamy do programu przy pomocy dyrektywy `#include` oraz pisząc w ostrych nawiasach nazwę zbioru z rozszerzeniem *.h*. Mówimy, że tego typu pliki nagłówkowe podlegają *prekompilacji*. Należy zwrócić uwagę, że najnowszy standard C++ z reguły już nie wymaga stosowania tego rozszerzenia i z powodzeniem możemy napisać np.:

```
#include <conio>
```

umożliwiając tym samym wykorzystywanie w naszych programach pewnych funkcji zdefiniowanych w pliku nagłówkowym *conio.h*.

Dyrektywa #pragma hdrstop

Przy pomocy dyrektywy `#pragma` jesteśmy w stanie przekazać kompilatorowi pewne dodatkowe informacje. Jeżeli po zakończeniu listy plików nagłówkowych użyjemy `#pragma hdrstop` (ang. *header stop*), poinformujemy kompilator, że właśnie wystąpił koniec listy plików nagłówkowych, które mają być prekompilowane.

Komentarz: Czy: kompilowane?

Dyrektywa #pragma argsused

Użycie tej dyrektywy zapobiega ewentualnemu wyświetlaniu komunikatu będącego ostrzeżeniem, że jeden z argumentów funkcji nie jest wykorzystywany. Dyrektywę `#pragma argsused` (ang. *arguments used*) należy umieszczać przed funkcją, tak jak pokazuje to rys. 2.2. W naszym programie przedstawionym na wydruku 2.1 zrezygnowaliśmy z tej dyrektywy z oczywistych względów.

Usunięto: jej

Usunięto: niej

Konsolidacja

Biblioteki właściwe zarówno C, jak C++ są opisane w definicjach, które zawierają jednocześnie wszystkie niezbędne funkcje wykorzystywane przy konstrukcji odpowiednich programów. W momencie, kiedy zostanie użyta jakaś funkcja nie będąca częścią programu (funkcje takie nazywamy bibliotecznymi), kompilator zapamięta jej nazwę. Kod wynikowy tekstu programu zostanie w odpowiedni sposób połączony z kodami istniejącymi w używanych bibliotekach. Proces tan określamy jako konsolidację lub linkowanie.

Usunięto: ich

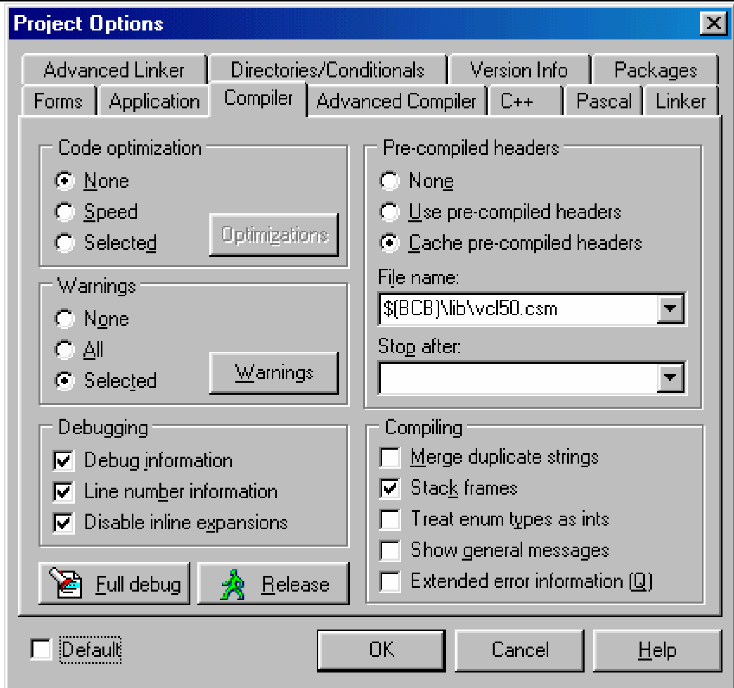
Usunięto: c

Konfigurujemy Opcje Projektu

Zanim zaczniemy na serio uruchamiać nasze programy i aplikacje, poświęćmy trochę uwagi kilku najważniejszym opcjom, przy pomocy których możemy skonfigurować nasz projekt. Zajrzyjmy do menu Project|Options...|Packages. Pierwszą, która się pojawi będzie pokazana na rysunku 2.3 zakładka Compiler:

Usunięto: niektórym

Usunięto: z jakimi



Rys. 2.3. Zakładka Compiler

Wciśnięty przycisk Full debug zapewni nam możliwość debuggowania programu w trakcie jego pisania lub sprawdzania. Stosowanie tej konfiguracji jest zalecane na etapie projektowania i testowania programów. Jeżeli natomiast dojdziemy do wniosku, że aplikacja nasza jest już w pełni gotowa i nie będzie wymagała dalszych ulepszeń (nie będziemy już więcej zaglądać do jej kodu), wystarczy wcisnąć Release. Włączona opcja Cache pre — compiled headers przyśpieszy włączanie do programu plików nagłówkowych, które muszą być poddane prekompilacji.

Posługując się opcjami dostępnymi w Advanced Compiler możemy m.in. ustalić typ procesora, na którym nasz program ma działać, rozmiar danych oraz czy program będzie kompilowany w standardach opisanych przez Borlanda, ANSI, System UNIX V lub Kernighana i Ritchie'go (K&R). Jeżeli nie mamy jakiś specjalnych wymagań, lepiej zbytnio nie ingerować w te opcje.

Bardzo ciekawą pozycją jest Runtime packages. Jeżeli pole Build with runtime package pozostanie zaznaczone (będzie aktywne), możemy mieć spore problemy z uruchomieniem naszego programu, o ile nie będzie znajdował się w instalacyjnym katalogu Buildera *\BIN*. Wynika to z faktu, że nasza aplikacja do prawidłowego działania potrzebować będzie paru dodatkowych bibliotek. W momencie, kiedy Build with runtime packages pozostanie odznaczone (nieaktywne), biblioteki te zostaną automatycznie dołączone do pliku wykonywalnego programu, zwiększając tym samym jego rozmiar[1]. Dla naszych potrzeb pole to pozostanie nieaktywne, tak jak pokazano na rysunku 2.4. Kiedy dołączać lub nie poszczególne biblioteki, każdy musi zadecydować sam. Jeżeli zależy nam na otrzymaniu pliku wykonywalnego o stosunkowo niewielkich rozmiarach, możemy je włączyć, należy jednak pamiętać, że wówczas w aktualnym katalogu razem z plikiem wykonawczym muszą znajdować się poszczególne biblioteki.

Usunięto: –

Komentarz: Czy: kompilacji?

Usunięto: m. in.

Usunięto: u

Usunięto: zie

Usunięto: m

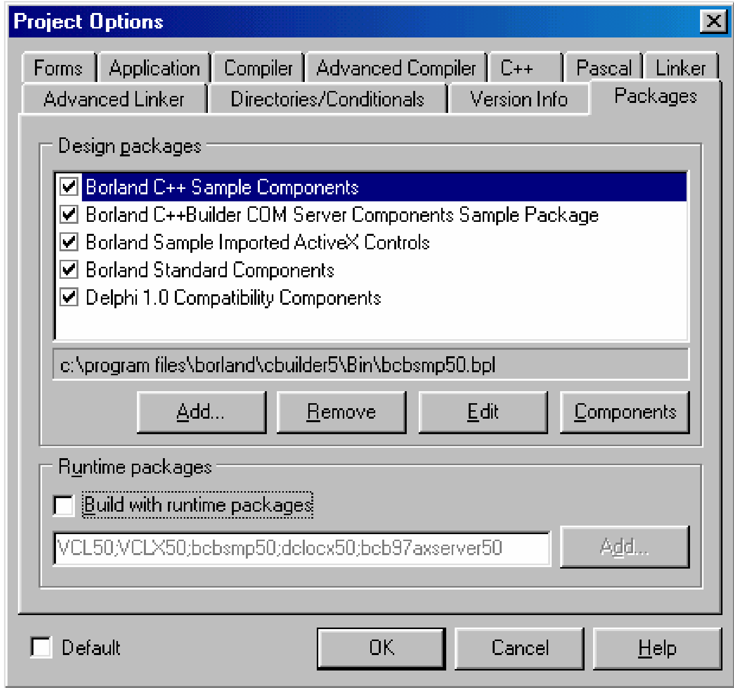
Usunięto: warto

Usunięto: a

Komentarz: Czy: wykonywalnym?

[1] W przypadku bardzo rozbudowanych pojedynczych aplikacji lub grup aplikacji zalecane jest dołączanie tych bibliotek głównie dla bezpieczeństwa funkcjonowania systemu Windows.

Rys. 2.4. Zakładka Packages



Przejdźmy z kolei do zakładki Linker. Jej wygląd pokazany jest na rys. 2.5. W panelu Linking znajduje się bardzo ciekawa opcja Use dynamic RTL. W przypadku, gdy pozostanie ona zaznaczona, nasz program wykonywalny może potrzebować do prawidłowego działania dwóch niewielkich zestawów procedur DLL: *borlndmm.dll* oraz *cc3250mt.dll*. Wymienione zestawy procedur DLL (ang. *Dynamic Link Library*) należą do grupy bibliotek RTL (ang. *Run- Time Libraries*). Wykorzystywane są podczas uruchamiania programów wykonawczych, ponadto te z przyrostkiem *mt* (ang. *Multi Thread*) wspomagają elementy wielowątkowego działania aplikacji i systemu operacyjnego. Dla naszych potrzeb opcja ta zostanie odznaczona, tzn. będziemy jawnie włączać je do naszych programów.

Należy jednak powiedzieć, że jawne włączanie do aplikacji zbyt wielu różnych bibliotek nigdy nie jest dobrym pomysłem. Programy wykonywalne nie powinny być zbyt duże, było to jedną z idei powstania bibliotek dołączanych dynamicznie. Czytelnik sam może się przekonać, że plik uruchomieniowy *bcb.exe* tak potężnego narzędzia, jakim jest C++Builder 5 ma rozmiar mniejszy niż 1 MB.

Komentarz: Czy to prawidłowa nazwa?

Usunięto: DLL-i:

Usunięto: DLL-e

Komentarz: Czy: wykonywalnych?

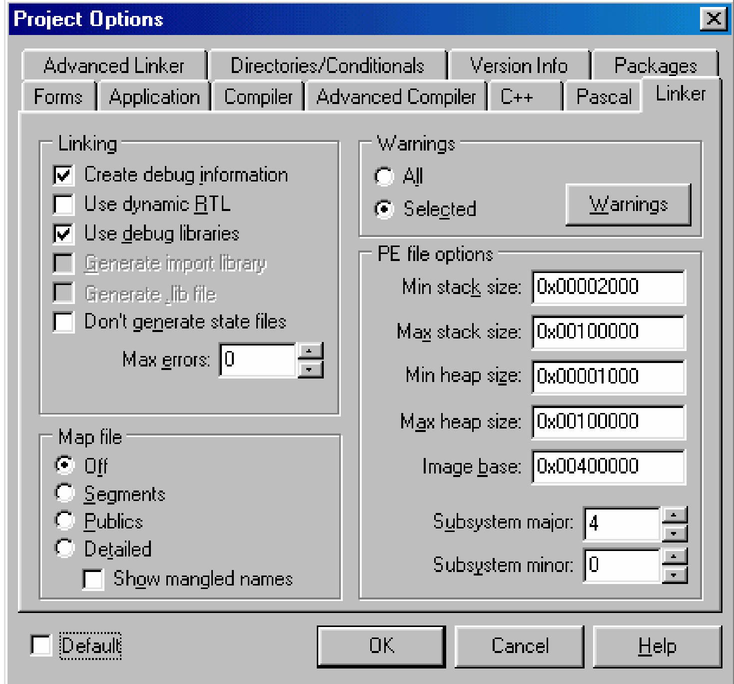
Komentarz: Która? Use dynamic RTL?

Komentarz: Odnaczona, tzn. nieaktywna, a jeśli tak, to nie będziemy jawnie włączać dynamicznych bibliotek do naszych programów.

Komentarz: Do czego odnosi się zaimek „je"? Do bibliotek?

Komentarz: Czy to poprawne określenie?

Rys. 2.5. Zakładka Linker z odznaczoną opcją Use dynamic RTL



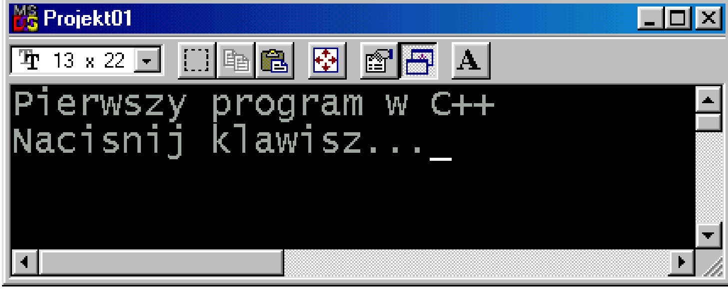
Korzystając z karty Application możemy nadać własny, unikalny tytuł projektowanej aplikacji, jak również zmienić jej ikonę, którą np. możemy wykonać sami, posługując się przedstawionym na rys. 1.34 edytorem graficznym.

Przy pomocy Version Info możemy kontrolować wersję programu. Kompilator będzie automatycznie podawać kolejny numer wersji po każdej kompilacji, pod warunkiem oczywiście, że zaznaczymy opcję Auto-increment build number. Ciekawostką jest również możliwość umieszczenia tu danych o autorze programu, jak i krótkiego opisu programu.

Uruchamiamy program

Teraz, kiedy dokonaliśmy właściwych ustawień opcji projektu, możemy skompilować i uruchomić projekt naszego modułu *Unit01.cpp*, zawierającego tekst źródłowy programu. Wystarczy w tym celu użyć opcji menu Run|Run (F9) lub prościej, z paska narzędzi wybierzmy przycisk Run (F9). Po uruchomieniu na ekranie powinniśmy zobaczyć okienko DOS, w którym wyświetlany jest napis będący efektem wykonania programu:

Rys. 2.6. *Projekt01.exe* w trakcie działania



Efekt działania programu na pewno nie jest czymś bardzo odkrywczym, niemniej jednak stanowić będzie dla nas pretekst do zapoznania się z pewnymi ważnymi pojęciami, których zrozumienie

okaże się niezbędne, jeżeli zechcemy w przyszłości projektować naprawdę dobrze działające aplikacje.

Zajrzyjmy do katalogu *\Projekt01*, powinno znajdować się w nim 6 plików:

- *Projekt01.exe*. Jest programem wykonywalnym (ang. *executable program*). Powstał on w wyniku działania konsolidatora łączącego standardowe funkcje biblioteki C++ z naszym kodem *Unit01.cpp*. Jeżeli odpowiednio skonfigurowaliśmy opcje projektu (tak jak na rysunkach 2.4 oraz 2.5) program ten można uruchamiać samodzielnie bez konieczności odwoływania się do innych plików.
- *Projekt01.bpr*. Zawiera wszystkie niezbędne instrukcje wykorzystywane przy tworzeniu projektu (ang. *builder project*)[2]. Jest tam opis samego projektu, opis opcji ustawień środowiska programisty IDE, opcji ustawień konsolidatora i wiele innych opisów. Zawartości tego pliku w żadnym wypadku nie należy modyfikować ręcznie, ani zmieniać jego nazwy w sposób dowolny, tzn. korzystamy jedynie z menu File|Save Project As... Pliki takie są przechowywane w formacie XML. Po uruchomieniu Buildera, kiedy chcemy poddać edycji nasz program, otwieramy go odwołując się do nazwy jego projektu poleceniem File|Open Project.
- *Projekt01.bpf*. Projekt pliku (ang. *borland project file*) utworzony w przypadku, gdy korzystamy ze środowiska C++Buildera, zaś programy piszemy w C lub C++, tak jak w naszym przykładzie.
- *Projekt01.tds*. (ang. *table debug symbols*). Plik binarny przechowujący informacje m.in. o włączonych bibliotekach i plikach nagłówkowych. Jest tworzony w momencie konsolidacji programu.
- *Unit01.cpp*. Jest tekstem źródłowym programu (ang. *source code*). Tekst źródłowy, który często bywa nazywany kodem, jest bezpośrednio wczytywany przez kompilator.
- *Unit01.obj*. Jest kodem wynikowym programu (ang. *object code*). Stanowi translację (przekład) tekstu źródłowego na język zrozumiały dla komputera. Kod wynikowy jest zawsze wczytywany przez konsolidator (linker).

Usunięto: ż

Usunięto: Są nimi

Komentarz: Czy to poprawne określenie?

Usunięto: ręcznie

Usunięto: nazwy jego

Usunięto: K

Usunięto: m. in.

Wszystkie wymienione pliki powinny znajdować się w katalogu, w którym zapisujemy projekt aplikacji. Utworzenie oddzielnego katalogu dla każdego z projektów bardzo ułatwia pracę z C++Builderem, w sposób znaczący ogranicza też możliwość przypadkowej utraty któregoś ze zbiorów. Należy zdawać sobie sprawę z faktu, że jeżeli utracimy np. plik projektu *.bpr*, aplikację będziemy musieli projektować praktycznie od początku.

Podsumowanie

Po przeczytaniu tego rozdziału powinniśmy się nieco oswoić ze środowiskiem programisty oferowanym przez Borland C++Builder 5. Wiemy już co to jest projekt, z jakich elementów się składa i jaka jest ich struktura. Umiemy też odpowiednio, według własnych potrzeb skonfigurować opcje projektu. Wiadomości te okażą się nam bardzo pomocne w dalszej części książki. Pokazaliśmy też, że korzystając ze środowiska BCB 5 możemy pisać konsolowe programy w „tradycyjnym” C++, a nawet w zwykłym C. Pewne dodatkowe elementy języka C++ zostaną przedstawione w następnym rozdziale.

Komentarz: Czy to poprawne sformułowanie?

Komentarz: Niestety, nie wiem jaki jest prawidłowy zapis skrótu BCB5.

[2] Dla naszych potrzeb nazwy wszystkich projektów zapisywanych na dysku będą nazwami polskimi.

Rozdział 3
Elementarz C++

W rozdziale tym krótko omówimy podstawowe typy danych, z jakimi możemy spotkać się pisząc programy w języku C++. Trochę miejsca poświęcimy instrukcjom sterującym, przypomnimy też sposób budowy i wykorzystania funkcji oraz struktur. Przypomnimy pojęcie wskaźnika i adresu.

Podstawowe typy danych oraz operatory arytmetyczne

Zarówno w języku C, jak i w języku C++ wyróżniamy pięć podstawowych typów danych:

`int` — typ całkowity. Używany jest do zapamiętywania i zapisywania liczb całkowitych.

Usunięto: –

`float` — typ zmiennopozycyjny (zmiennoprzecinkowy).

Usunięto: –

Usunięto: ;

`double` — typ zmiennoprzecinkowy podwójnej długości. Zmienne typu `float` oraz `double` umożliwiają zapamiętywanie i zapisywanie liczb rzeczywistych, posiadających część całkowitą i ułamkową. Część ułamkową oddzielamy kropką.

Usunięto: –

`char` — typ znakowy. Typ ten stosujemy do zapamiętywania i zapisywania znaków ASCII oraz krótkich liczb reprezentowanych na 8 bitach.

Usunięto: –

`void` — typ pusty. Wykorzystywany bywa w następujących sytuacjach. Po pierwsze, za jego pomocą możemy deklarować funkcje nie zwracające żadnych wartości. Po drugie, możemy deklarować funkcje, które nie pobierają argumentów. Po trzecie, umożliwia on tworzenie ogólnych wskaźników.

Usunięto: –

Usunięto: korzystając z niego

Usunięto: deklarując

Usunięto: ce

> Każda zmienna użyta w programie musi być najpierw zadeklarowana, to znaczy należy poinformować kompilator z jakiego typu danymi przyjdzie mu pracować. Właśnie na tej podstawie dokonuje się sprawdzania poprawności rezultatu wykonania danej operacji arytmetycznej lub logicznej. Zmienne globalne można deklarować bądź przed wywołaniem głównej funkcji `main()`, bądź w jej ciele.

Komentarz: ?

Jako przykład wykorzystania w programie jednego z opisanych wyżej typów danych niech nam posłuży prosty algorytm przedstawiony na wydruku 3.1.

Usunięto: tych

Wydruk 3.1. Algorytm realizujący operację dodawania dwóch liczb typu `float`.

```
#include <iostream.h>
#include <conio.h>
#pragma hdrstop
```

```
float x, y, z; // deklaracja zmiennych

int main()
{
  cout << endl << "Podaj dwie liczby " << endl;
  cin >> x >> y;
  z = x + y;
  cout << x << " + " << y <<" = " << z;
  cout << endl << "Naciśnij klawisz...";
  getch();
  return 0;
}
```

W tym prostym przykładzie wykorzystaliśmy operatory dodawania + oraz instrukcję przypisania =.
Spośród innych operatorów arytmetycznych należy wyróżnić:

Operator	Działanie	Postać matematyczna
-	odejmowanie	z = x - y;
*	mnożenie	z = x * y;
/	dzielenie	z = x / y;
%	dzielenie modulo	z = x % y;
--	zmniejszanie o jeden (dekrementacja)	z = z - 1;
++	zwiększanie o jeden (inkrementacja)	z = z + 1;
+=	skrót przypisania	z += x; to samo co: z = z + x;
-=	skrót odejmowania	z -= x; z = z - x;
*=	skrót mnożenia	z *= x; z = z * x;
/=	skrót dzielenia	z /= x; z = z / x;

Usunięto: ¶

Ogólna postać instrukcji przypisania wygląda następująco:

```
Zmienna = wyrażenie;
```

Można też stosować wielokrotnie takie instrukcje, np.:

```
z = x = y = 0;
```

Jedyną i najlepszą metodą zapoznania się z właściwościami zmiennych oraz ze sposobami użycia niektórych operatorów arytmetycznych jest wykonanie paru ćwiczeń.

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 3.1.

Po uruchomieniu C++Buildera wybierz File|New|Console Wizard. Opcję Console Wizard skonfiguruj podobnie jak na rys. 2.1.

Usunięto: W

Usunięto: S

1. Posługując się kodem pokazanym na wydruku 3.1, spróbuj przetestować działanie programu.

Usunięto: S

2. Zmodyfikuj program w ten sposób, by przetestować działanie omówionych operatorów arytmetycznych.

3. Oblicz wartość wyrażenia `(x + y)*y /(y - x)`. Podobnie jak na wydruku 3.1, wyświetl w postaci komunikatu, jakie działania były kolejno wykonywane.

Usunięto: W

4. Sprawdź rezultat działania programu z różnym wykorzystaniem operatorów dekrementacji oraz inkrementacji, np.:

```
cout << x << " + " << y << " = " << z++;
```

oraz

```
cout << x << " + " << y << " = " << ++z;
```

Operatory relacyjne i logiczne

Każda różna od zera liczba, z którą spotykamy się w C, posiada wartość `TRUE` (prawda) , natomiast liczba 0 posiada wartość `FALSE` (nieprawda). Wyrażenia, w których występują operatory relacyjne bądź logiczne, zwracają wartość 1 (`TRUE`) lub 0, czyli `FALSE`. W zależności od potrzeb posługujemy się następującymi operatorami:

Operatory relacyjne

<table>
<tr><th>Operator</th><th>Działanie</th></tr>
<tr><td>></td><td>większy</td></tr>
<tr><td><</td><td>mniejszy</td></tr>
<tr><td>>=</td><td>większy lub równy</td></tr>
<tr><td><=</td><td>mniejszy bądź równy</td></tr>
<tr><td>==</td><td>równy</td></tr>
<tr><td>!=</td><td>różny</td></tr>
</table>

Operatory logiczne

<table>
<tr><th>Operator</th><th>Działanie</th></tr>
<tr><td>&&</td><td>koniunkcja AND (i)</td></tr>
<tr><td>||</td><td>alternatywa OR (lub)</td></tr>
<tr><td>!</td><td>negacja NOT (nie)</td></tr>
</table>

Posługując się przedstawionymi operatorami należy zawsze pamiętać, że posiadają one różny priorytet wykonywania kolejnych działań. Rozpatrzmy to na przykładzie wyrażenia `5-4 != 0`. Jego wartość obliczana jest w taki sposób, że najpierw zostanie wykonana operacja odejmowania liczb, a dopiero potem sprawdzony warunek, czy rezultat odejmowania jest różny od zera, tzn.: `(5-4) != 0`. Należy też pamiętać, że operator `()` ma największy priorytet. Jeżeli nie jesteśmy pewni priorytetów stosowanych operatorów zawsze w wątpliwych sytuacjach możemy posłużyć się właśnie operatorem `()`.

Deklarowanie tablic

Tablice służą do zapamiętywania danych tego samego typu, i, podobnie jak zmienne, wymagają przed użyciem deklaracji. Deklarując tablicę informujemy nasz komputer o potrzebie przydzielenia odpowiedniej ilości pamięci oraz o kolejności rozmieszczenia elementów tablicy. W najprostszy sposób tablicę zawierającą 10 liczb całkowitych deklarujemy następująco:

```
int Tablica[10];
```

Dla komputera oznaczać to będzie potrzebę zarezerwowania 10 kolejnych pól pamięci dla 10 liczb całkowitych typu `int`. Każda taka liczba będzie zapamiętana na 4 bajtach. Deklarując tablice, np. `Tablica[n]` należy pamiętać, że w C++ poszczególne ich elementy są ponumerowane za pomocą indeksu od `0` do `n-1`. W naszym przypadku kolejnymi elementami tablicy będą: `Tablica[0]`, `Tablica[1]`, ... `Tablica[9]`. W bardzo prosty sposób przypisujemy wartości elementom tablic, np.:

```
Tablica[5] = 25;
```

W analogiczny sposób deklarujemy tablice znakowe. Jeżeli zapiszemy:

```
char znak[20];
```

Oznaczać to będzie, że zarezerwowaliśmy w pamięci 20 8-bitowych pól, w których będą przechowywane dane typu `char`. Do takiej tablicy również możemy wpisać łańcuch znaków:

```
char napis[20] = "Borland C++Builder 5";
```

lub, co jest równoważne:

```
char napis[11] = {'B','o','r','l','a','n','d',' ','C','+','+'};
```

Mimo, iż napis "Borland C++Builder 5" składa się z 20 znaków (spacje też są traktowane jako znaki), to musieliśmy zadeklarować tablicę składającą się również z 20 elementów, a nie 19 (pamiętamy, że indeksy liczymy od 0). Wynika to z faktu, że C++ posługuje się łańcuchami znakowymi zakończonymi znakiem '\0' `NUL` (ASCII 00). Jeżeli taki napis zechcemy wyświetlić wystarczy napisać:

```
cout << endl << napis;
```

Tablice mogą być jednowymiarowe (tzw. wektory) lub wielowymiarowe. Jeżeli zechcemy zadeklarować dwuwymiarową tablicę, składającą się z 10 elementów typu `float`, możemy napisać:

Usunięto: a

```
float Tablica [2][5];
```

Oznaczać to będzie następujące ponumerowanie jej indeksów:

Usunięto: Co o

```
Tablica[0][0], Tablica[0][1], Tablica[0][2], Tablica[0][3],
Tablica[0][4]
Tablica[1][0], Tablica[1][1], Tablica[2][2], Tablica[2][3],
Tablica[2][4]
```

Elementom takich tablic również można przypisywać wartości. Na przykład:

```
float Tablica[2][3] = {{1,2,3}, {4,5,6.5}};
```

Oznaczać to będzie przypisanie jej indeksom następujących wartości:

Usunięto: Co o

```
Tablica[0][0]=1; Tablica[0][1]=2; Tablica[0][2]=3;
Tablica[1][0]=4; Tablica[1][1]=5; Tablica[1][2]=6.5;
```

Elementy takich tablic wyświetlamy w sposób bardzo prosty:

```
cout << endl << Tablica[1][1];
```

Instrukcje sterujące

W C oraz C++ zdefiniowane są trzy kategorie instrukcji sterujących:

- Instrukcje warunkowe, niekiedy nazywane instrukcjami wyboru, czyli `if` oraz `switch`.
- Instrukcje iterakcyjne, zwane też instrukcjami pętli, lub po prostu pętlami. Należą do nich `for`, `while` oraz `do...while`.
- Instrukcje skoku: `break`, `continue`, `goto`.

Instrukcja `return` jest też zaliczana do instrukcji skoku, z tego powodu, iż wykonanie jej wpływa na przebieg wykonywania funkcji lub programu jako całości.

Instrukcja if

W ogólnym przypadku blok instrukcji `if` przyjmuje następującą postać:

```
if (wyrażenie)
  {
    ciąg instrukcji
  }
    else {
      ciąg instrukcji
   }
```

Zastosowanie jej rozpatrzmy na przykładzie prostego programu wykonującego operację dzielenia. Operacje takie w pewnych wypadkach mogą być trochę niebezpieczne dla naszego algorytmu, gdyż jak zapewne wiemy niedopuszczalne jest wykonywanie dzielenia przez zero.

Usunięto: jest

Wydruk. 3.2. Program obrazujący ideę posługiwania się blokiem instrukcji `if`

```
#include <iostream.h>
#include <conio.h>
#pragma hdrstop

void main()
{
  float x, y, z;

  cout << endl << "Podaj dwie liczby " << endl;
  cin >> x >> y;

  if ((x - y) != 0)
    {
     z = (x + y)/(x - y);
    }
    else
      cout << endl << "Uwaga! Próba dzielenia przez zero";

  cout << x << " + " << y <<" = " << z;
  cout << endl << "Naciśnij klawisz...";
  getch();
}
```

Ćwiczenie do samodzielnego wykonania

Ćwiczenie 3.2.

Wykorzystując jako ściągawkę kod programu przedstawionego na wydruku 3.2, sprawdź rezultat jego wykonania z innymi operatorami relacji.

Usunięto: S

Komentarz: Czy chodzi o rezultat wykonania kodu przy zastosowaniu innych operatorów relacyjnych?

Instrukcja switch

Decyzyjna instrukcja `switch` (niekiedy nazywana instrukcją przesiewu) porównuje kolejno wartości wyrażenia, które musi być typu całkowitego, znakowego lub wyliczeniowego z listą liczb całkowitych, lub innych stałych znakowych.

```
switch( wyrażenie typu całkowitego int, znakowego char lub enum ) {
  case stała1:
      lista instrukcji;
      break;
  case stała2:
      lista instrukcji;
      break;
   ...
   default:
      lista instrukcji;
}
```

Jako przykład praktycznego wykorzystania omawianej instrukcji niech nam posłuży poniższy algorytm.

Wydruk. 3.3. Sposób użycia w programie instrukcji decyzyjnej `switch`.

```
#include <iostream.h>
#include <conio.h>
#pragma hdrstop

int x = 3, y, z;

int main()
{
  cout << endl << "Podaj liczbę całkowitą z przedziału <0,3>" << endl;
  cin >> y;
  switch(z = x - y) {
  case 0:
     cout << " Wprowadzono liczbę 3";
     break;
  case 1:
     cout << " Wprowadzono liczbę 2";
     break;
  case 2:
     cout << " Wprowadzono liczbę 1";
     break;
  case 3:
     cout << " Wprowadzono liczbę 0";
     break;
  default:
     cout << " Nieprawidłowa liczba ";
  }
  cout << endl << "Naciśnij klawisz...";
  getch();
  return false;
}
```

Po uruchomieniu programu wpisujemy jakąś liczbę, która będzie przechowywana w zmiennej `y`. Następnie zostanie wykonana operacja odejmowania wprowadzonej liczby od liczby `3`, zadeklarowanej w programie i przechowywanej w zmiennej `x`. Wynik działania zostanie przypisany zmiennej `z`. Następnie nastąpi cykl sprawdzający, jaką liczbą jest rezultat odejmowania. Instrukcja `default` będzie wykonana, wtedy, gdy nie będzie można znaleźć wartości zgodnej z wartością wyrażenia podanego w `switch`.

Usunięto: występuje

Usunięto: a

Usunięto: w tedy

Ćwiczenie do samodzielnego wykonania

Ćwiczenie 3.3.

Postaraj się zaprojektować algorytm rozróżniający wprowadzane z klawiatury znaki. Jako przykład niech nam posłuży poniższy szkielet programu:

```
...
char znak;
int main()
{
```

```
  cout << endl << " Wprowadź znak z klawiatury" << endl;
  cin >> znak;
  switch(znak) {
  case 'a':
     cout << " Wprowadzono literę a";
     break;
...
}
```

Instrukcja for

Każde współczesne środowisko programistyczne udostępnia nam możliwość wykonywania ciągu instrukcji aż do spełnienia założonego warunku. W instrukcji for warunek taki określany jest mianem warunku predefiniowanego.
W ogólnej postaci instrukcja for składa się z trzech głównych części:

Komentarz: Czy to poprawne określenie?

```
for(inicjalizacja; predefiniowany warunek; inkrementacja)
{
 grupa instrukcji;
}
```

Instrukcje tego typu posługują się z reguły tzw. zmiennymi sterującymi (licznikiem wykonań). W części inicjalizującej zmiennej sterującej zostanie nadana wartość początkowa. Całość instrukcji będzie wykonywana do czasu spełnienia predefiniowanego warunku. Sposób modyfikacji zmiennej sterującej po każdorazowym zakończeniu danego cyklu jest zdefiniowany w części inkrementacyjnej.

Komentarz: Czy chodzi o to, że „W części początkowej wykonywania instrukcji, zmiennej sterującej zostanie nadana pewna wartość.”?

Komentarz: Danego cyklu czego?

Komentarz: ? W części zwiększania o 1?

Instrukcja for nie może być zakończona średnikiem. Znak ; określa koniec wykonywanych instrukcji. Każda instrukcja for zakończona średnikiem zostanie wykonana najwyżej jeden raz.

Usunięto: wykona się

Sposób wykorzystania w programie wymienionej instrukcji pomoże nam zilustrować przykład programu cyklicznie wyświetlającego kwadraty oraz pierwiastki kwadratowe liczb całkowitych z przedziału <1; 10>.

Usunięto: co

Usunięto: rz

Wydruk 3.4. Idea posługiwania się instrukcją for.

```
#include <iostream.h>
#include <conio.h>
#pragma hdrstop

double i, j, k;

int main()
{
  for(i = 1; i <= 10; i++)
    {
     j = pow(i, 2);
     k = sqrt(i);
     cout << endl << "kwadrat" << i <<"= " << j << "pierwiastek="<< k;
    }
  cout << endl << "Naciśnij klawisz...";
  getch();
  return 0;
}
```

W celu obliczenia pierwiastka liczby użyliśmy funkcji sqrt(), której rezultat musi być liczbą zmiennoprzecinkową, np. double. Do obliczania kwadratu liczby wykorzystana została funkcja pow(), której ogólna definicja brzmi:

```
double pow(double x, double y);
```

Matematyczny zapis tej funkcji jest bardzo prosty: x^y.
Oczywiście funkcję tę z powodzeniem można użyć również do obliczania pierwiastka kwadratowego: `pow(x, 0.5)`, co oznacza $\sqrt{x}$, lub jakichkolwiek innych potęg.

Usunięto: ą

Ćwiczenie do samodzielnego wykonania

Ćwiczenie 3.4.

W pętli `for` oblicz i wyświetl sumę oraz różnicę trzecich potęg czterech różnych liczb całkowitych. Zakres zmienności tych liczb ustalmy od 1 do 20.

Komentarz: Czy określenie „pętla for” jest równoznaczne z określeniem „instrukcja for”?

Usunięto: O

Usunięto: W

Nieskończona pętla for

Ciekawą własnością języka C, zaadoptowaną również do języka C++, jest możliwość wykorzystania w programie pętli nieskończonej w postaci `for(;;)`, tzn. nie sprawdza się tutaj żadnych warunków kontynuacji. Aby zakończyć wykonywanie takiej pętli, należy w odpowiednim miejscu programu użyć instrukcji `break`. Poniższy przykład ilustruje to zagadnienie.

Usunięto: która została oczywiście

Usunięto: a

Wydruk.3.5. Nieskończona pętla `for`.

```
#include <iostream.h>
#include <conio.h>
#pragma hdrstop

double i = 1, j;
int main()
{
  for(;;)
    {
      j = pow(i, 2);
      cout << endl << "kwadrat " << i <<" = " << j;
      i++;
      if (j >= 1000)
         break;
    }
  cout << endl << "Naciśnij klawisz...";
  getch();
  return 0;
}
```

Instrukcja while

Instrukcja iterakcyjna `while` przybiera następującą postać:

```
while(warunek)
   {
      instrukcja lub grupa instrukcji;
   }
```

Powyższa pętla będzie wykonywana tak długo, dopóki *warunek* nie będzie spełniony, przy czym jego prawdziwość sprawdzana jest przed wykonaniem *grupy instrukcji*. Kiedy *warunek* przybierze wartość `FALSE`, działanie programu będzie kontynuowane od pierwszej instrukcji znajdującej się za pętlą. Poniższy przykład pomoże nam zrozumieć mechanizm działania pętli `while`.

Wydruk. 3.6. Kwadraty liczb całkowitych obliczane w pętli `while`.

```
#include <iostream.h>
#include <conio.h>
#pragma hdrstop

double i = 1, j;

int main()
{
  while ( i <= 10)
    {
      j = pow(i, 2);
      cout << endl << "kwadrat " << i <<" = " << j;
      i++;
    }
  cout << endl << "Naciśnij klawisz...";
  getch();
  return 0;
}
```

Ćwiczenie do samodzielnego wykonania

Ćwiczenie 3.5.

Zmodyfikuj pokazany na powyższym wydruku program w ten sposób, by cyklicznie wyświetlał wprowadzane z klawiatury znaki, aż do momentu wybrania litery `'a'`. W tym celu można posłużyć się funkcją `getchar()`.

Instrukcja do...while

Istnieje zasadnicza różnica pomiędzy instrukcjami `for`, `while` oraz `do...while`. O ile w przypadku `for` oraz `while` warunek wykonywania instrukcji sprawdzany jest już na początku, to w przypadku `do...while` sprawdza się go na końcu. Z faktu tego wynika, że instrukcje znajdujące się w pętli `do...while` będą wykonane co najmniej jeden raz. Pętla ta w ogólnej postaci wygląda następująco:

```
do{
    sekwencja instrukcji;
}while(warunek);
```

Zamieszczony poniżej przykład programu, wczytującego dowolne znaki wprowadzane z klawiatury, pomoże nam zrozumieć zasadę działania pętli `do...while`, która będzie wykonywana do momentu wprowadzenia małej lub dużej litery `'x'`.

Wydruk 3.7. Zasada działania instrukcji powtarzającej `do...while`.

```
#include <iostream.h>
#include <conio.h>
#pragma hdrstop

char znak;
int main()
{
  do
    {
      znak = getche();
      cout << endl <<"Wczytano znak " << znak << endl;
  } while (znak != 'x' && znak != 'X');
  cout << endl << "Naciśnij klawisz...";
  getch();
```

```
    return 0;
  }
```

Ćwiczenie do samodzielnego wykonania

Ćwiczenie 3.6.

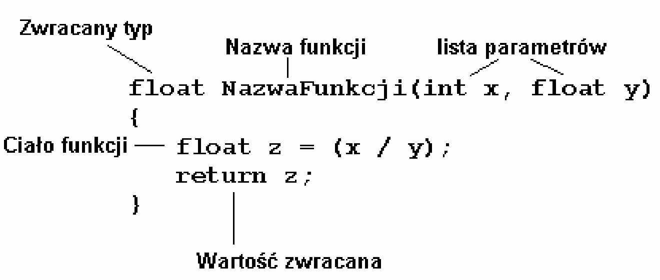
Korzystając z powyższego przykładu, Zbuduj algorytm obliczający i wyświetlający na ekranie pierwiastki trzeciego stopnia całkowitych liczb nieujemnych z przedziału od 20 do 50.

Funkcje w C++

Jednym z najważniejszych elementów zarówno języka C, jak i języka C++ są funkcje. Wiemy już, że każdy pisany przez nas program musi zawierać przynajmniej jedną funkcję — funkcję `main()`. Poznaliśmy też już parę funkcji bibliotecznych oferowanych w standardzie ANSI C. Były nimi: `getch()`, `getchar()`, `getche()`, `pow()` czy chociażby `sqrt()`. W celu odróżnienia funkcji od zmiennych, po nazwie funkcji piszemy nawiasy okrągłe. Funkcje są najprostszym sposobem ujęcia pewnych obliczeń, działań czy innych operacji w jednym elemencie strukturalnym, do którego możemy odwoływać się wielokrotnie w trakcie programu.

Z punktu widzenia budowy funkcji, każda z nich składa się z następujących elementów:

Usunięto: –
Usunięto: W
Usunięto: międzyczasie
Usunięto: p
Usunięto: ,
Usunięto: b
Usunięto: Z anatomicznego punktu widzenia
Usunięto: funkcja



Rys. 3.1. Budowa funkcji w C++

Najlepszą metodą zapoznania się ze sposobami użycia funkcji w programie jest stworzenie odpowiedniego algorytmu. Pamiętamy, że dotychczas w celu obliczenia potęgi jakiejś liczby wykorzystywaliśmy biblioteczną funkcję `pow()`. Zbudujmy teraz samodzielnie jej prosty odpowiednik. Nasza funkcja, nazwijmy ją *power* (potęga), będzie obliczała wartości kolejnych całkowitych potęg liczby 2.

Usunięto: ,
Usunięto: n

Wydruk 3.8. Program korzystający z funkcji `power()` w celu obliczania kolejnych potęg liczby 2.

```
#include <iostream.h>
#include <conio.h>
#pragma hdrstop

int power(int x, int y);    // prototyp funkcji
int i;
int main()
{
  for( i = 1; i <= 10; i++)
     cout << endl << power(2,i); // wywołanie funkcji w głównym
                                 // programie

  cout << endl << "Naciśnij klawisz...";
```

```
  getch();
  return 0;
}

int power(int x, int y)       // definicja funkcji power (potęga)
{
   int z = 1, i;
   for(i = 1; i <= y; i++)
      z = z * x;
   return z;
}
```

Każda funkcja, samodzielnie przez nas napisana, przed użyciem musi być odpowiednio zadeklarowana w programie. Deklarujemy ją przed główną funkcją `main()`. Działanie to określa się mianem podania prototypu funkcji wraz z jej parametrami formalnymi. Parametry formalne są takimi parametrami, z jakimi funkcja jest zadeklarowana. W naszym przykładzie parametrami takimi są dane typu `int x` oraz `y`. Następnie treść naszej funkcji umieszczamy za głównym programem. Samo wywołanie funkcji `power()`, już z parametrami aktualnymi, następuje w treści głównej funkcji `main()`. Parametrami aktualnymi nazywamy dane, z jakimi funkcję wywołujemy.

Usunięto: ,

Istnieją dwa sposoby dołączenia własnej funkcji do programu. Jeżeli treść funkcji zdecydujemy się umieścić za głównym programem, należy podać jej prototyp. Jeżeli treść funkcji umieszczamy bezpośrednio przed główną funkcją `main()` podawanie prototypu nie jest wymagane.

Wielką zaletą posługiwania się funkcjami jest to, że możemy do nich odwoływać się wielokrotnie z możliwością podawania za każdym razem innych parametrów aktualnych, np.:

```
cout << endl << power(2,i) << " " << power (3,i) << " " << power(4,i);
```

W językach C oraz C++ wywoływana funkcja na ogół nie zmienia wartości zmiennych w funkcjach wywołujących. Mówimy, że tego rodzaju funkcje przekazują swe argumenty przez wartość. Jeżeli zachodzi potrzeba, by funkcja zmieniała wartości zmiennych w funkcji wywołującej, to ta ostatnia musi przekazać adres zmiennej, zaś funkcja wywoływana musi zadeklarować odpowiedni argument jako wskaźnik.

Usunięto: W ogólności w

Ćwiczenie do samodzielnego wykonania

Ćwiczenie 3.7.

Zaprojektuj program, który będzie pełnić rolę najprostszego kalkulatora. Wszystkie podstawowe działania, takie jak: dodawanie, odejmowanie, mnożenie, dzielenie czy obliczanie odwrotności liczb umieść w odpowiednich funkcjach. Funkcje te należy wywoływać w głównym programie z odpowiednimi parametrami aktualnymi.

Usunięto: U

Wskazania i adresy

Zarówno w języku C, jak i C++ istnieją dwa bardzo ważne pojęcia, którymi często posługujemy się pisząc programy. Są nimi wskazanie i adres. Wskazaniem nazywamy dane identyfikujące pewien obiekt, którym może być np. zmienna lub funkcja. Adresem nazywamy pewien atrybut danej wskazującej lokalizujący jej miejsce w pamięci. W ogólnym wypadku powiemy, że wskaźnik jest zmienną, która zawiera adres innej zmiennej w pamięci komputera. Istnieją dwa bardzo ważne operatory umożliwiające nam posługiwanie się adresami obiektów.

Usunięto: ą

Usunięto: ą

Komentarz: ?

Jednoargumentowy operator `&` (zwany operatorem adresowym lub referencji) podaje adres obiektu. Jeżeli zapiszemy:

```
px = &x;
```

oznaczać to będzie, że wskaźnikowi px przypisaliśmy adres zmiennej x. Powiemy, że px wskazuje na zmienną x, lub że px jest wskaźnikiem do zmiennej x.

Z kolei operator * (gwiazdka) zwany operatorem wyłuskiwania w działaniu na zmienną spowoduje, że zmienna taka będzie traktowana jako adres danego obiektu. Operator ten traktuje swój argument jako adres obiektu i używa tego adresu do pobrania zawartości obiektu.

Komentarz: ?

Rozpatrzmy dwie grupy instrukcji wykonujących podobnie wyglądające przypisania:

```
y  = x;
```

oraz

```
px = &x;
y = *px;
```

O pierwszym przypisaniu powiemy, że wykonując je nadajemy zmiennej y dotychczasową wartość zmiennej x. O drugim zaś powiemy, że najpierw wskaźnikowi px przypisaliśmy adres zmiennej x, a następnie zmiennej y nadaliśmy dotychczasową wartość zmiennej, której adres wskazuje wskaźnik px. Widzimy więc, że te dwie grupy instrukcji wykonują dokładnie to samo. Zrozumienie idei użycia w programie wskazań i adresów ułatwi nam poniższy wydruk.

Usunięto: z nich

Wydruk 3.9. Program obliczający kolejne potęgi liczby 2. Wywołując funkcję power() korzystamy ze wskaźnika do danej i, która jest potęgą liczby 2, a zarazem parametrem aktualnym funkcji.

Komentarz: Czy chodzi o zmienną i? Nie można użyć słowa „dana” w znaczeniu rzeczownikowym, a tylko przymiotnikowym, np. dana książka lub dana zmienna.

Usunięto: będącą kolejną

```
#include <iostream.h>
#include <conio.h>
#pragma hdrstop

int power(int x, int *y);   // prototyp funkcji
int i;
int main()
{
  for( i = 1; i <= 10; i++)
     cout << endl << power(2, &i);

  cout << endl << "Naciśnij klawisz...";
  getch();
  return 0;
}

int power(int x, int *y)       // zapis funkcji power (potęga)
{
   int z = 1;
   int i;
   for(i = 1; i <= *y; i++)
      z = z * x;
   return z;
}
```

Funkcja power(int x, int *y) będzie zmieniać wartość jednego ze swoich argumentów całkowitych. Jeżeli zechcemy, by w momencie wywołania przekazywać argumenty przez adres, zmienna (lub zmienne) przekazywana funkcji power() musi być poprzedzona operatorem &: power(2, &i); tylko wówczas będzie utworzony odpowiedni wskaźnik.

Usunięto: ,

Struktury

Strukturę tworzy zbiór zmiennych, złożony z jednej lub z logicznie powiązanych kilku zmiennych różnych typów, zgrupowanych pod jedną nazwą. Najprostszym przykładem wykorzystania struktur mogą być wszelkiego rodzaju listy płac pracowników, czy chociażby dane związane z

ewidencją ludności. Struktury stosujemy po to, by ułatwić sobie zorganizowanie pracy z większą ilością skomplikowanych danych. Podobnie jak każdy typ danych, również i struktura wymaga deklaracji w programie. Istnieje kilka sposobów deklaracji struktury. Na potrzeby naszej książki przedstawimy jeden z najprostszych. Aby logicznie pogrupować dane różnych typów stosujemy struktury deklarowane przy pomocy słowa kluczowego `struct`. Następnie podajemy nazwę struktury określając w ten sposób jej typ. W nawiasach klamrowych deklarujemy elementy składowe struktury (często zwane polami). Na koniec należy z reguły podać listę nazw struktur określonego typu, z których będziemy w przyszłości korzystać.

Jako przykład zadeklarujmy strukturę typu `Student`, w której elementach będziemy przechowywać pewne dane związane z osobami wybranych studentów.

Usunięto: y

```
struct Student       // deklaracja ogólnego typu struktury Student
  {
   char Imie[20];
   char Nazwisko[20];
   float EgzaminMatematyka;
   float EgzaminFizyka;
   float EgzaminInformatyka;
   char JakiStudent[30];
  };
```

Tak więc w kolejnych polach przechowywać będziemy imię i nazwisko danej osoby, oceny z egzaminów wybranych przedmiotów i na koniec naszą opinię o studencie.

Następnie zainicjujmy dwie struktury statyczne typu `Student` pod nazwami `Student1` oraz `Student2`:

```
static struct Student
Student1 = {"Wacek", "Jankowski", 5, 5, 5, "bardzo dobry student"};
static struct Student
Student2 = {"Janek", "Wackowski", 2.5, 2.5, 2.5, "kiepski student"};
```

Jeżeli zechcemy, aby struktura zajmowała stale ten sam obszar pamięci oraz, aby była dostępna z każdego miejsca programu, należy zadeklarować ją jako statyczną — `static`.

Usunięto: –

Oczywiście w ten sam sposób możemy utworzyć jeszcze szereg innych struktur danego typu, co zilustrowane jest na przykładzie algorytmu pokazanego na wydruku 3.9.

Wydruk 3.10. Przykład wykorzystania informacji zawartych w strukturach.

Usunięto: 9.

```
#include <iostream.h>
#include <conio.h>
#pragma hdrstop

int main()
{
  struct Student       // deklaracja ogólnego typu struktury Student
  {
   char Imie[20];
   char Nazwisko[20];
   float EgzaminMatematyka;
   float EgzaminFizyka;
   float EgzaminInformatyka;
   char JakiStudent[30];
  };

  static struct Student
  Student1 = {"Wacek", "Jankowski", 5, 5, 5, "bardzo dobry student"};
  static struct Student
  Student2 = {"Janek", "Wackowski", 2.5, 2.5, 2.5, "kiepski student"};

  struct Student S3, S4;
  S3 = Student2;
  S3.EgzaminFizyka = Student1.EgzaminFizyka;
  S4 = Student1;
```

```
  cout << endl << S3.Imie <<" "<< S3.Nazwisko <<" "<<
         S3.EgzaminFizyka<<" "<< S3.EgzaminMatematyka<<" "<<
         S3.EgzaminInformatyka;

  cout << endl << S4.JakiStudent;

  cout << endl << "Naciśnij klawisz...";
  getch();
  return 0;
}
```

Analizując powyższy wydruk na pewno zauważymy, że aby przekazać wartość pojedynczego pola numerycznego struktury `Student1` do struktury `S3` wykonaliśmy przypisanie:

```
S3.EgzaminFizyka = Student1.EgzaminFizyka;
```

Stąd wniosek, że po to, by odwołać się do danego elementu struktury, należy podać jej nazwę i po kropce wybrany element (pole) struktury. Jeżeli zechcemy przekazać zawartość całej struktury do innej tego samego typu, wykonujemy normalne przypisanie podając ich nazwy:

```
S3 = Student2;
```

Widzimy więc, że struktury pomagają zorganizować sobie pracę z danymi różnych typów. Wówczas grupa związanych ze sobą zmiennych może być traktowana jako jeden obiekt. Zagadnienie struktur w C++ jest niezwykle bogate. Z racji charakteru książki pominięte zostały takie pojęcia, jak tablice struktur, wskaźniki do struktur czy opis struktur odwołujących się do samych siebie. Pojęcia takie wprowadza się na zaawansowanym kursie programowania, zatem zainteresowanych Czytelników odsyłam do bogatej literatury przedmiotu. Z powodzeniem można też skorzystać z niezwykle bogatych w te treści plików pomocy C++Buildera 5.

Ćwiczenie do samodzielnego wykonania

Ćwiczenie 3.8.

Zaprojektuj program, przy pomocy którego będziesz mógł przechowywać wszystkie interesujące informacje o swoich znajomych (adres, nr telefonu, e-mail, itp.).

Usunięto: B

Podsumowanie

W niniejszym rozdziale zostały przedstawione podstawowe pojęcia związane z programowaniem w C++. Omówiliśmy podstawowe typy danych, operatory arytmetyczne i logiczne, tablice, instrukcje sterujące przebiegiem działania programu, funkcje oraz struktury. Przypomnienie wiadomości na temat wskazań i adresów bardzo nam w przyszłości pomoże w zrozumieniu mechanizmu obsługi zdarzeń już z poziomu Borland C++Builder 5. Przedstawienie szeregu pożytecznych przykładów praktycznego wykorzystania elementów języka C++ ułatwi nam samodzielne wykonanie zamieszczonych w tym rozdziale ćwiczeń.

Rozdział 4
Projektowanie obiektowe OOD

Projektowanie obiektowe (ang. *object-oriented design*) stanowi zespół metod i sposobów pozwalających elementom składowym aplikacji stać się odpowiednikiem obiektu lub klasy obiektów rzeczywiście istniejących w otaczającym nas świecie. Wszystkie aplikacje budujemy po to, by odzwierciedlały lub modelowały rzeczywistość, która nas otacza. Aplikacje takie będą zbiorem współdziałających ze sobą różnych elementów. Przed rozpoczęciem tworzenia takiego programu należy zastanowić się, jakie cele ma on spełniać i przy pomocy jakich elementów (obiektów) cele te będą realizowane. Należy zatem:

- Zdefiniować nowy (lub zaimplementować istniejący) typ danych — klasę.
- Zdefiniować obiekty oraz ich atrybuty.
- Zaprojektować operacje, jakie każdy z obiektów ma wykonywać.
- Ustalić zasady widoczności obiektów.
- Ustalić zasady współdziałania obiektów.
- Zaimplementować każdy z obiektów na potrzeby działania aplikacji.
- Ustalić mechanizm dziedziczenia obiektów.

Usunięto: –

Klasa

Definiuje nowy typ danych, będących w istocie połączeniem danych i instrukcji, które wykonują na nich działania umożliwiając tworzenie (lub wykorzystanie istniejących) obiektów będących reprezentantami klasy. Jedna klasa może być źródłem definicji innych klas pochodnych.

Komentarz: Na kim? Na czym?

Obiekt

Stanowi element rzeczywistości, którą charakteryzuje pewien stan. Każdemu obiektowi zawsze można przypisać określony zbiór metod, czyli operacji. Klasa jest również obiektem.

Metody

Każdy wykorzystywany w programie obiekt wykonuje (lub my wykonujemy na nim) pewne czynności — operacje, potocznie zwane metodami. Metodami nazywamy funkcje (lub procedury) będące elementami klasy i obsługujące obiekt przynależny do danej klasy.

Usunięto: –

Widoczność obiektów

Jeżeli uznamy to za konieczne, możemy ustalić zakres widoczności obiektów w odniesieniu do fragmentu programu. Obiekt taki będzie korzystał ze zmiennych dostępnych jedynie dla metod klasy, w której je zdefiniowano.

Współdziałanie obiektów

Jeżeli obiekt lub grupę obiektów uczynimy widocznymi w całej aplikacji, należy ustalić zasady porozumiewania się obiektów, czyli relacje pomiędzy nimi. Dla każdego z obiektów ustalamy ściśle określony zbiór reguł i funkcji, dzięki którym korzystać z niego mogą inne obiekty.

Implementacja obiektu

Implementacja, czyli oprogramowanie obiektu, oznacza stworzenie kodu źródłowego obsługującego metody z nim związane. W szczególności korzystając z zasad programowania obiektowo - zdarzeniowego, z poszczególnymi obiektami kojarzymy odpowiadające im zdarzenia.

Usunięto: –

Zdarzenie

Zdarzenie (ang. *event*) określane jest jako zmiana występująca w aktualnym stanie obiektu, będąca źródłem odpowiednich komunikatów przekazywanych do aplikacji lub bezpośrednio do systemu. Reakcja obiektu na wystąpienie zdarzenia udostępniana jest aplikacji poprzez funkcję obsługi zdarzeń (ang. *event function*) będącą wydzieloną częścią kodu.

Dziedziczenie

Jest mechanizmem programowania obiektowego. Pozwala na przekazywanie właściwości klas bazowych klasom pochodnym (potomnym). Nowe klasy będą dziedziczyć, czyli przejmować z klas, będących ich przodkami, pola, metody, instrukcje i właściwości.

Programowanie zorientowane obiektowo

Zapoznamy się teraz z jednym ze sposobów błyskawicznego zaprojektowania i stworzenia aplikacji w środowisku C++ Builder 5 posługując się techniką programowania zorientowanego obiektowo. W tym celu zbudujemy naprawdę prosty program, którego zadaniem będzie wyświetlenie w odpowiednim miejscu formularza znanego nam już napisu. Wykorzystując polecenie menu File|New|Application stwórzmy na pulpicie nowy formularz.

- Korzystając z karty właściwości inspektora obiektów w jego cechę `Caption` (opis) wpiszmy `Projekt02`. Cechę `Name` (nazwa) pozostawmy nie zmienioną jako `Form1`[3]. Poleceniem File|Save As... zapiszmy główny moduł projektu w katalogu *\Projekt02\Unit02.cpp*.
- Następnie poprzez File|Save Project As... w tym samym katalogu zapiszmy sam projekt jako *Projekt02.bpr*. W ten prosty sposób przygotowaliśmy nasz formularz do dalszych działań.

[3] W tym oraz dalszych przykładach pozostaniemy przy nazwie Form1, po to by niepotrzebnie nie komplikować dalszych rozważań. Jeżeli byśmy zmienili tę nazwę, zmieni się ona również np. w pliku *Unit02.h*

Klasa TForm1

Formularz jest pierwszym obiektem, z którym spotykamy się rozpoczynając pisanie aplikacji. Jeżeli moduł tworzonego obecnie projektu zapisaliśmy jako *\Projekt02\Unit02.cpp*, to w tym samym katalogu C++Builder powinien wygenerować plik nagłówkowy *Unit02.h*. Zerknijmy do jego wnętrza. Zostawmy na boku dyrektywy preprocesora, natomiast przypatrzmy się dokładnie definicji tworzącej klasę naszego formularza:

Wydruk 4.1. Zawartość modułu *Unit02.h*.

```
#ifndef 02H
#define 02H
//-------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
//-------------------------------------------------------------------
class TForm1 : public TForm
{
__published:      // IDE-managed Components
private:           // User declarations
public:           // User declarations
        __fastcall TForm1(TComponent* Owner);
};
//-------------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//-------------------------------------------------------------------
#endif
```

Właśnie otrzymaliśmy automatycznie wygenerowaną przez BCB (skrót od Borland C++ Builder) definicję przykładowej klasy, na bazie której będą w przyszłości tworzone obiekty. BCB oferuje nam słowo kluczowe `class` pozwalające na tworzenie obiektów. Przed stworzeniem obiektu określamy jego ogólną postać korzystając właśnie ze słowa `class`. Klasa `TForm1` dziedziczy własności klasy `TForm`, będącej bazową klasą formularza. Definicja klasy składa się z kilku części. W sekcji `__published` umieszczane będą deklaracje[4] funkcji, czyli deklaracje metod związanych z komponentami pochodzącymi z biblioteki VCL. Sekcja `private` przeznaczona jest dla zmiennych (zwanych tutaj polami) oraz metod widzianych tylko wewnątrz klasy. W sekcji `public` deklarować można pola i metody mogące być udostępniane innym.

Usunięto: a

Komentarz: ?

Zauważmy, że C++Builder umożliwia też tworzenie aplikacji nie zawierających formularza (por. *Projekt01.exe*, który był aplikacją konsolową), zatem klasa `TForm1` nie miała tam zastosowania. Z faktu tego wynika brak pliku *Unit01.h* w wymienionym projekcie.

Konstruktor TForm1()

Zanim zaczniemy na serio korzystać z obiektu naszego formularza musi on zostać odpowiednio zainicjowany. Dokonuje się to poprzez specjalną funkcję składową, noszącą taką samą nazwę jak klasa, do której należy. Prototyp takiej funkcji (nazywanej konstruktorem) z parametrami wygląda następująco:

Usunięto: tego

Komentarz: Co należy? Obiekt czy funkcja?

```
__fastcall TForm1(TComponent* Owner);
```

[4] W C++ deklaracje funkcji nazywane bywają często ich prototypami.

Ponieważ konstruktor nie zwraca żadnej wartości, nie określa się jego typu (przez domniemanie jest on typu nieokreślonego, czyli `void`). Konwencja `__fastcall` (szybkie wywołanie) zapewnia, że parametry konstruktora zostaną przekazane poprzez rejestry procesora. Dodatkowo zapis konstruktora z parametrem `Owner` informuje, że właścicielem (ang. *owner*) wszystkich komponentów jest `TComponent` mówi nam, że `TComponent` jest wspólnym przodkiem dla wszystkich komponentów z biblioteki VCL włącznie ze stworzoną klasą `TForm1`. Klasa `Tcomponent`, wprowadzając wiele metod i właściwości, umożliwia m.in. obsługę komponentów z poziomu inspektora obiektów. Pełny tekst konstruktora klasy `TForm1` zostanie automatycznie umieszczony w module *Unit02.cpp*, tam też zostanie zainicjowany.

Komentarz: ?

Usunięto: m. in.

Usunięto: on

Formularz jako zmienna obiektowa

Jak się zapewne domyślamy, projekt naszej aplikacji będzie składał się nie tylko z formularza, ale również z modułów i innych zasobów. Wszystkie części składowe aplikacji przechowywane są w odpowiednich plikach, w większości wypadków tworzonych automatycznie przez BCB. Ponieważ C++Builder 5 (podobnie jak C i C++) pozwala na konsolidację oddzielnie skompilowanych modułów dużego programu, musi zatem istnieć jakiś sposób na poinformowanie wszystkich plików wchodzących w skład projektu o występowaniu zmiennych globalnych (widocznych w całej aplikacji), niezbędnych w danym programie. Najlepszym sposobem by to osiągnąć, jest zadeklarowanie zmiennych globalnych tylko w jednym pliku i wprowadzenie deklaracji przy pomocy specyfikatora `extern PACKAGE` (ang. *zewnętrzny pakiet*) w innych plikach (zob. wydruk 4.1).

Usunięto: używając

Nasz formularz jest obiektem, lub jak kto woli zmienną obiektową, której deklaracja zostanie umieszczona w głównym module formularza *Unit02.cpp*:

```
#include <vcl.h>
#pragma hdrstop
#include "Unit02.h"
...

TForm1 *Form1;
...
```

Widzimy więc, że nazwa klasy stała się nowym specyfikatorem typu danych.

Tworzymy aplikację

Po tych być może trochę długich, ale moim zdaniem bardzo ważnych wyjaśnieniach, zobaczmy jak w praktyce posługiwać się klasą `TForm1` i w jaki sposób uzupełnić ją o szereg obiektów.

Usunięto: ujemy

Usunięto: możemy

Pisząc programy w środowisku BCB z reguły korzystamy z biblioteki VCL. Chociaż do jej omawiania przejdziemy dopiero w następnych rozdziałach, nic nie stoi na przeszkodzie, abyśmy już teraz powoli zaczęli oswajać się z jej elementami. Jest to również dobry moment by zapoznać się z jeszcze paroma właściwościami inspektora obiektów.

Pierwsza aplikacja

Ćwiczenie 4.1.

1. Ustalmy na początek rozmiary formularza. Cechę `Height` (wysokość) ustalmy, powiedzmy, na 300 pikseli, zaś cechę `Width` (szerokość) na 480.
2. Rozwińmy cechę `Constraints` (ograniczenie). W odpowiednie miejsca wpiszmy wartości pokazane na rys. 4.1.

Rys. 4.1. Ograniczenie rozmiarów formularza

⊟ Constraints	(TSizeConstrain
MaxHeight	300
MaxWidth	480
MinHeight	300
MinWidth	480

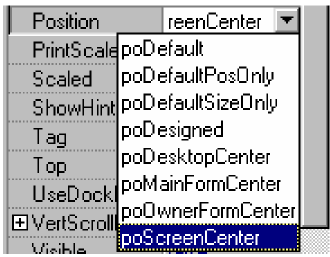
Przypisując poszczególnym cechom wartości zgodne z uprzednio ustalonymi rozmiarami formularza, uzyskamy taki efekt, że w momencie uruchomienia aplikacji formularz nie będzie „rozpływał" się po ekranie nawet po kliknięciu na pole maksymalizacji.

Usunięto: taki

Usunięto: w

3. Przejdźmy do cechy `Position` i wybierzmy np. `poScreenCenter` (rys. 4.2).

Rys. 4.2. Cecha Position inspektora obiektów



Spośród widocznych opcji (które możemy samodzielnie przetestować), wybrana przez nas sprawi że w momencie uruchomienia aplikacji formularz pozostanie w centrum ekranu (ale nie pulpitu). Jeżeli oczywiście w inspektorze obiektów cechy `Align` (zakotwiczenie) nie ustawiliśmy inaczej, niż na `alNone`.

Usunięto: zapewni

Usunięto: jej

Usunięto: nie ustawiliśmy inaczej jak na `alNone`

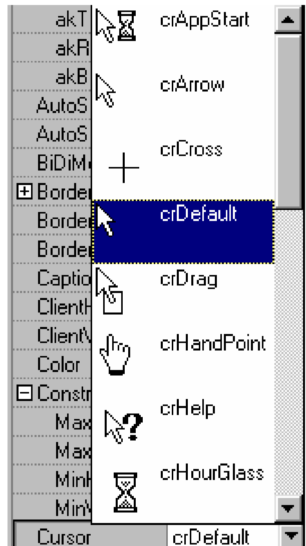
Usunięto: .

Komentarz: Dostosować sposób wyróznienia do słowa „`Align`"

Usunięto: ,

4. Na koniec, należy zadbać o postać kursora, jaki będzie obowiązywać w obszarze formularza. Standardowym kursorem jest `crDefault` (domyślny). Jeżeli pojawia nam się jakiś inny kursor (i jeżeli nam nie odpowiada) w obszarze formularza, rozwińmy cechę `Cursor` inspektora obiektów i wybierzmy kursor domyślny (rys. 4.3). Oczywiście, w zależności od upodobań, każdy może wybrać ten najbardziej mu odpowiadający.

Rys. 4.3. Rodzaje dostępnych kursorów

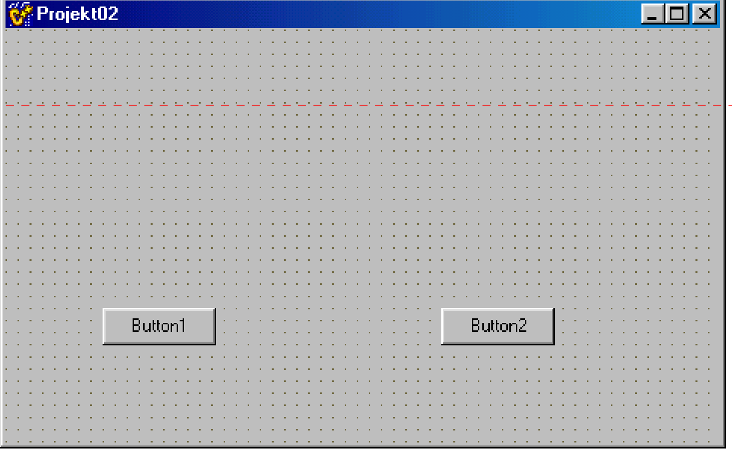


Skoro posiadamy już pewne wiadomości na temat obiektowej natury formularza oraz ustaliliśmy wstępnie jego parametry, to na tak przygotowanej formie możemy już umieścić odpowiednie komponenty. Ponieważ jedynym zadaniem naszego programu będzie wyświetlenie w odpowiedni sposób napisu analogicznego jak na rys. 2.6, posłużymy się dwoma komponentami `TButton` z karty Standard. Aby je przenieść do obszaru formy należy kliknąć komponent z podpowiedzią `Button`, a następnie również klikając, ale już obszar formularza, umieścić go w odpowiednim miejscu. Forma naszego projektu powinna wyglądać tak jak na rys. 4.4.

Usunięto: na
Usunięto: w
Usunięto: ze

Rys. **4.4.** Sposób rozmieszczenia komponentów TButton na obszarze formularza

Usunięto: w



Korzystając z inspektora obiektów oraz z karty właściwości — Properties, cechę `Caption` przycisku `Button2` zmień na &Zamknij. Podobnie cechę `Caption` przycisku `Button1` zmień na &Tekst. Cechy `Name` pozostawimy nie zmienione jako `Button1` oraz `Button2`. Oczywiście, żeby zmienić cechy tych przycisków, należy najpierw je zaznaczyć, tylko raz klikając odpowiedni komponent. Znak `&`, który występuje w nazwach przycisków oznacza, że litera, stojąca bezpośrednio po nim, będzie stanowić klawisz szybkiego dostępu (szybkiego wywołania) do funkcji obsługi odpowiedniego zdarzenia. Również przy pomocy inspektora obiektów możemy zmienić cechy `Font` obu przycisków, dobierając najbardziej odpowiadający nam rodzaj czcionki.

Usunięto: –
Usunięto: na
Komentarz: ?
Usunięto: ich
Usunięto: tym samym rodzaj najbardziej odpowiadającej nam czcionki.

Funkcja obsługi zdarzenia

Przyciski umieszczone na formularzu wyglądają na pewno bardzo ładnie, niemniej jednak muszą jeszcze spełniać określoną rolę w naszej aplikacji, mianowicie należy uczynić je zdolnymi do generowania zdarzeń. Dla każdego z nich należy stworzyć funkcję obsługi zdarzenia. Zacznijmy od przycisku zamykającego aplikację. Klikając dwukrotnie przycisk Zamknij, dostaniemy się do wnętrza odpowiedniej funkcji obsługi zdarzenia `Button2Click()`.

Usunięto: na

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{

}
```

Jednak zanim cokolwiek tam wpiszemy, przeanalizujmy pokrótce powyższe zapisy. Już teraz zaglądając do pliku *Unit02.h* zobaczymy, że w deklaracji klasy `TForm1` występuje prototyp funkcji (metody) `Button2Click()`, która od tej pory stała się funkcją składową klasy. W miejscu, w którym występuje pełen tekst źródłowy funkcji składowej klasy (w naszym wypadku w pliku głównego modułu formularza *Unit02.cpp*), kompilator musi być poinformowany, do której klasy wywoływana funkcja należy. Dokonujemy tego posługując się operatorem `::`.

Komentarz: Od jakiej pory?

Użycie operatora rozróżnienia zakresu `::` (ang. *scope resolution operator*) informuje

kompilator, że przykładowa funkcja `Button2Click()` należy do przykładowej klasy `TForm1`. Ponieważ klasy C++Buildera mogą zawierać wiele funkcji (metod) o takich samych nazwach, należy poinformować kompilator o tym, że w danej chwili któraś z nich może być wykorzystywana. W tym celu stosuje się nazwę klasy wraz z operatorem rozróżnienia zakresu: `TForm1::`.

Usunięto: któraś z nich.

Do w miarę pełnego opisu konstrukcji funkcji obsługi przykładowego zdarzenia potrzeba jeszcze wyjaśnić rolę jej parametrów. Z zapisu:

```
TObject *Sender
```

odczytamy: `*Sender` jest wskaźnikiem i wskazuje na dane typu `TObject`. `Sender` reprezentuje tutaj pewną właściwość polegającą na tym, że każdy obiekt z listy palety komponentów VCL musi być w pewien sposób poinformowany o tym, że będzie przypisana mu funkcja obsługi zdarzenia.

`TObject` jest bezwzględnym przodkiem wszystkich komponentów i obiektów VCL. Umieszczony jest na samym szczycie hierarchii obiektów VCL. W C++Builder 5 wszystkie egzemplarze obiektów mają postać 32-bitowych wskaźników do przydzielonej na stosie pamięci.

Poniżej przedstawiona funkcja obsługi zdarzenia zapewnia, że po uruchomieniu, aplikacja w stosownym dla nas momencie może być bezpiecznie zamknięta w odpowiedzi na wywołanie tej funkcji, czyli naciśnięcie odpowiedniego przycisku:

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{
   Application->Terminate();
}
```

Każdy program BCB zawiera zmienną globalną `Application` typu `TApplication`, która deklarowana jest następująco:

```
__fastcall virtual TApplication(Classes::TComponent* AOwner);
extern PACKAGE TApplication* Application;
```

W czasie tworzenia nowego projektu C++Builder konstruuje obiekt aplikacji i przypisuje mu właśnie zmienną `Application`. Obiekty klasy `TApplication` przechowują zasady współpracy aplikacji z systemem operacyjnym Windows, takie jak rozpoczynanie i kończenie aplikacji, tworzenie okna głównego itp. Właściwości i zdarzenia wymienionej klasy nie są dostępne z poziomu inspektora obiektów, natomiast właściwości aplikacji możemy zmieniać za pomocą opcji menu Project|Options...|Forms lub Application.

Istnieje wiele metod klasy `TApplication`, jedną z nich: `Terminate()`, którą właśnie przećwiczyliśmy. Jej pełna deklaracja wygląda następująco:

```
void __fastcall Terminate(void);
```

Funkcja ta umożliwia zamknięcie aplikacji. `Terminate()` nie jest oczywiście jedyną z prostszych w użyciu metod, które oferuje `TApplication`. Następną, równie prostą i użyteczną jest funkcja:

```
void __fastcall Minimize(void);
```

którą każdy może wypróbować już samodzielnie.

Należy oczywiście pamiętać, że do olbrzymiej większości metod przynależnych do odpowiednich klas odwołujemy się poprzez operator `->`.

Operatory `.` (kropka) i `->` (strzałka) wykorzystywane są do uzyskiwania dostępu do pojedynczych elementów zbiorczych typów danych, np. struktur i unii, do których jako całości odwołujemy się poprzez podanie ich nazwy. Kropkę wykorzystujemy w

przypadku wykonywania działań na tych obiektach, strzałkę zaś podczas korzystania ze wskaźników do tych typów danych.

Pod względem składni (zob. wydruk 4.1) klasa przypomina strukturę, ale różni się od niej tym, że oprócz obiektów może zawierać też funkcje, tzn. wiąże strukturę danych i możliwe do wykonania operacje na tej strukturze danych.

Przejdźmy teraz do zaprojektowania funkcji obsługi zdarzenia dla przycisku `Button1`, który nazwaliśmy Tekst. Aby wyświetlić odpowiedni napis na formularzu, zastosujemy najprostszą metodę, mianowicie wykorzystamy fakt, ze każdy formularz, będący w istocie pewnym komponentem, posiada swoje własne płótno (ang. *canvas*), reprezentowane przez klasę `TCanvas` posiadającą właściwość `Canvas` . Płótno stanowi obszar, na którym możemy wykonywać bardzo wiele operacji graficznych. Funkcja obsługi zdarzenia `Button1Click()` skojarzona z przyciskiem `Button1` może wyglądać jak poniżej:

Usunięto: w

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
     Canvas->Font->Style = TFontStyles() << fsBold << fsUnderline;
     Canvas->Brush->Color = clBtnFace;
     Canvas->Font->Color = clBlack;
     Canvas->Font->Height = 30;
     Canvas->TextOut(30,30, "Pierwsza aplikacja w C++Builder 5");
}
```

Widzimy, że sposób odwoływania się do obszaru płótna poprzez właściwość `Canvas` klasy `TCanvas` nie jest skomplikowany i nie powinien przedstawiać nam żadnych trudności. Wykorzystaliśmy tutaj kilka właściwości płótna, takich jak: czcionka (`Font`) i pędzel (`Brush`) oraz metodę `TextOut()` klasy `TCanvas`. Wykorzystując zagnieżdżenie obiektów odwołaliśmy się także do ich poszczególnych własności, takich jak: kolor (`Color`), styl (`Style`) oraz wysokość (`Height`). Funkcja:

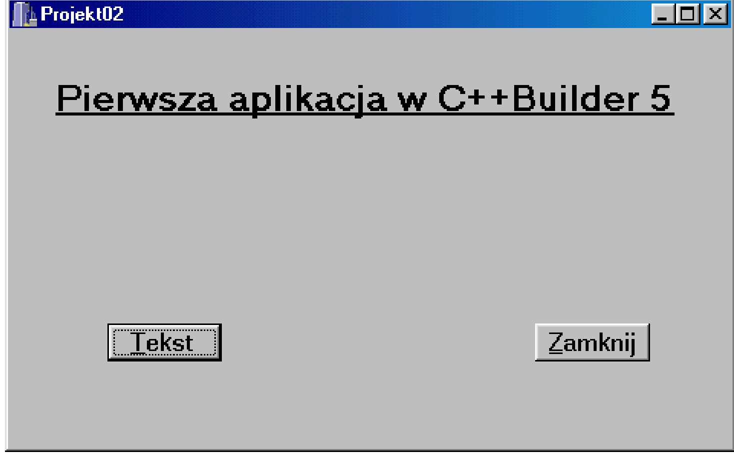
```
void __fastcall TextOut(int X, int Y, const AnsiString Text);
```

pozwala na umieszczenie dowolnego tekstu identyfikowanego przez stałą `Text` w miejscu formularza o współrzędnych `X`, `Y` Odległość liczona jest w pikselach. Lewy górny róg formularza ma współrzędne 0, 0. Nasza aplikacja po uruchomieniu powinna wyglądać podobnie jak na rys. 4.5.

Usunięto: P

Usunięto: (o

Usunięto:)

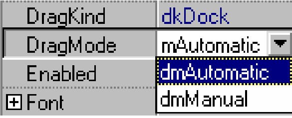


Rys. 4.5. *Projekt02.bpr* po uruchomieniu

Skoro tak dobrze nam idzie, to wypróbujmy jeszcze jeden komponent z karty Standard, mianowicie komponent edycyjny typu `TEdit`, który jak już powinniśmy się domyślać będzie

reprezentowany właśnie przez okienko o nazwie `Edit1`. Po wstawieniu go do formularza ustalmy jego rozmiar oraz typ czcionki (właściwość `Font` w inspektorze obiektów). Wykonajmy ponadto jeszcze dwie bardzo ciekawe czynności. Mianowicie cechy `DragKind` (rodzaj przemieszczania) oraz `DragMode` (tryb przemieszczania) ustalmy tak, jak pokazuje to rysunek 4.6.

Rys. 4.6.
Właściwości DragKind oraz DragMode inspektora obiektów



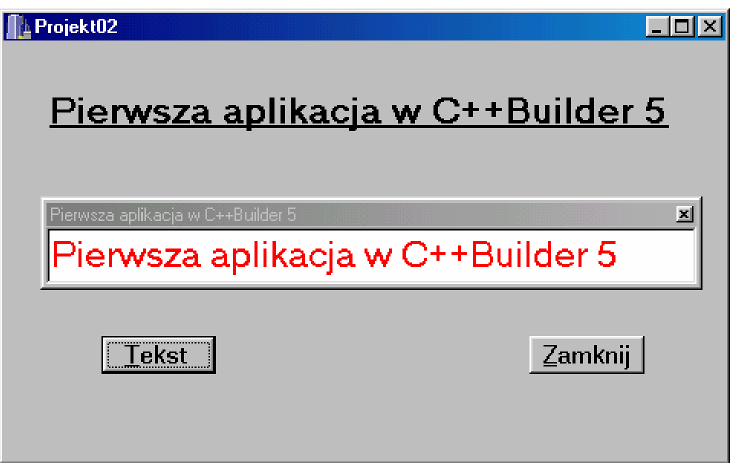
Ponadto, funkcję obsługi zdarzenia `Button2Click()` uzupełnijmy o niewielki fragment kodu:

```
//-- umieszczamy tekst w oknie edycji Edit1 --
     Edit1->Font->Color = clRed;
     Edit1->Text = "Pierwsza aplikacja w C++Builder 5";
```

Widzimy, że oprócz znanych już nam właściwości `Font` i `Color` użyliśmy jeszcze jednej — `Text`. Takie przypisanie ciągu znaków ujętych w cudzysłowy spowoduje, że tekst ten zostanie wyświetlony w oknie edycji, do którego cechy `Text` jest przypisany. Uruchommy aplikację i od razu kliknijmy obszar edycji `Edit1`, potem zaś przycisk Tekst. Wygląd formularza działającej aplikacji pokazany jest na rys. 4.7.

Usunięto: –
Komentarz: ?
Usunięto: w
Usunięto: na

Rys. 4.7.
Zmodyfikowany *Projekt02.bpr* po uruchomieniu



Dzięki odpowiednim ustawieniom, dokonanym przy pomocy inspektora obiektów w odniesieniu do komponentu `Edit1`, mamy możliwość swobodnego przesuwania go po całym ekranie, a także dowolnego zmieniania jego rozmiarów. Już na takich prostych przykładach możemy poznać potęgę programowania zorientowanego obiektowo. Kilka ruchów myszką przy sprowadzaniu komponentów w odpowiednie miejsce formularza, parę linijek kodu i trochę pracy z inspektorem obiektów, a efekt jest naprawdę dobry.

Na pewno też zauważymy, że przesuwając okienko po ekranie, w pewnym momencie zaczniemy zamazywać tekst wyświetlany na płótnie formularza. Nasze okno działa jak gumka do ścierania. Cóż, trzeba się z tym liczyć — nawet sama nazwa `Canvas` (płótno) sugeruje, że wszystko co umieszczamy na formularzu, korzystając z właściwości i metod płótna, nie będzie zbyt trwałe. Właśnie z tego powodu komponenty edycyjne odgrywają tak dużą rolę w bibliotece VCL. Poniżej zamieszczony został kompletny kod głównego modułu naszej aplikacji.

Usunięto: –
Usunięto: w

Wydruk 4.2 Kod głównego modułu *Unit02.cpp* projektu *Projekt02.bpr*

Usunięto: y

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

```
#include "Unit02.h"
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//--------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
   // konstruktor TForm1()
}
//--------------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
     Canvas->Font->Style = TFontStyles() << fsBold << fsUnderline;
     Canvas->Brush->Color = clBtnFace;
     Canvas->Font->Color = clBlack;
     Canvas->Font->Height = 30;
     Canvas->TextOut(30,30, "Pierwsza aplikacja w C++Builder 5");

    //-- wyświetlamy tekst w oknie edycji Edit
     Edit1->Font->Color = clRed;
     Edit1->Text = "Pierwsza aplikacja w C++Builder 5";
}
//--------------------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{
   Application->Terminate();
}
//-------------------------------------------------------------------
```

Przedstawiony powyżej bardzo prosty algorytm ma jednak pewną wadę, mianowicie jeżeli raz zamkniemy `Edit1` (dzięki jego własnemu polu zamknięcia), to już nie będziemy mieli możliwości, by w trakcie działania aplikacji odzyskać je z powrotem. Możemy zapobiec takiej sytuacji, projektując chociażby funkcje obsługi dwóch nowych zdarzeń, uruchamianych powiedzmy poprzez dwa nowe przyciski typu `TButton` z wykorzystaniem metod `Hide()` (ukryj) i `Show()` (pokaż):

Komentarz: Tzn. co?

```
//------ukrywa okno edycji Edit1--------------------------------------
void __fastcall TForm1::Button3Click(TObject *Sender)
{
    Edit1->Hide();
}
//-------przywraca okno edycji Edit1----------------------------------
void __fastcall TForm1::Button4Click(TObject *Sender)
{
    Edit1->Show();
}
//--------------------------------------------------------------------
```

Powyższe zapisy powinny nam wyjaśnić jeszcze jedną bardzo ważną rzecz, mianowicie w jaki sposób należy odwoływać się do obiektów oraz ich właściwości i metod w funkcjach obsługi zdarzeń związanych z zupełnie innymi obiektami.

Na zakończenie tego fragmentu naszych rozważań zauważmy, że funkcje obsługi zdarzeń budowane w oparciu o komponenty biblioteki VCL nie zwracają wartości powrotnej poprzez instrukcję `return`.

Ogólna postać aplikacji w C++Builder 5

Jak już wcześniej wspominaliśmy, wszystkie składniki naszej aplikacji przechowywane są w plikach. Zaglądając do katalogu *\Projekt02* przyjrzyjmy się, z jakiego rodzaju plikami mamy do czynienia:

- Znany nam już wykonywalny plik wynikowy *Projekt02.exe*. Jest utworzonym przez nas programem.

- Plik główny projektu *Projekt02.bpr*. O jego roli w naszej aplikacji już wcześniej wspominaliśmy.
- *Projekt02.tds* — *table debug symbols*, również powinien być już nam znany.
- Kod wynikowy aplikacji, czyli plik *Projekt02.obj*.
- Plik zasobów *Projekt02.res*. Jest binarnym plikiem zasobów (ang. *resources*). Zawiera m. in. ikonę.
- Plik główny naszej aplikacji, czyli *Projekt02.cpp*. Zawiera funkcję `WinMain()`.

Usunięto: -

Wydruk 4.3. Zawartość pliku z funkcją `WinMain()`.

```
#include <vcl.h>
#pragma hdrstop
USERES("Projekt02.res");
USEFORM("Unit02.cpp", Form1);
//---------------------------------------------------------------------
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
        try
        {
                Application->Initialize();
                Application->CreateForm(__classid(TForm1), &Form1);
                Application->Run();
        }
        catch (Exception &exception)
        {
                Application->ShowException(&exception);
        }
        return 0;
}
```

Programy pisane w Borland C++Builderze i posługujące się klasą formularza nie zawierają funkcji `main()`.Wszystkie pisane przez nas aplikacje rozpoczynają działanie od wywołania innej funkcji, mianowicie `WinMain()`, wywoływanej zgodnie z zasadami `WINAPI`, co jest wyraźnie zaznaczone w jej definicji. Otrzymuje ona wartość czterech parametrów. Pierwsze dwa, typu `HINSTANCE` (w wolnym tłumaczeniu określane jako uchwyty lub jak kto woli identyfikatory przypadku) są niezbędne z prostego powodu, mianowicie Windows w obecnym kształcie jest systemem wielozadaniowym, w związku z tym w danej chwili może działać jednocześnie wiele egzemplarzy tego samego programu. Parametry przypisane typom `HINSTANCE` określają aktualnie działające egzemplarze programu. Parametr typu `LPSTR` jest wskaźnikiem do łańcucha znaków zawierającego argumenty wiersza poleceń, które są określane w trakcie uruchamiania aplikacji. Ostatni parametr typu całkowitego `int` określa sposób wyświetlania okna formularza po rozpoczęciu działania aplikacji. Proces inicjacji — metoda `Initialize()`, tworzenia formularza — metoda `CreateForm()` oraz uruchamiania aplikacji — metoda `Run()` rozgrywa się pomiędzy klauzulami `try...catch` (w wolnym tłumaczeniu: próbuj...przechwyć, złap). Jeżeli proces ten nie powiedzie się, na ekranie ujrzymy stosowny komunikat w postaci wygenerowanego przez system tzw. wyjątku (ang. *exception*), wyświetlanego przy pomocy funkcji `ShowException()`.

Usunięto: inicjalizacji

Usunięto: –

Usunięto: –

Usunięto: –

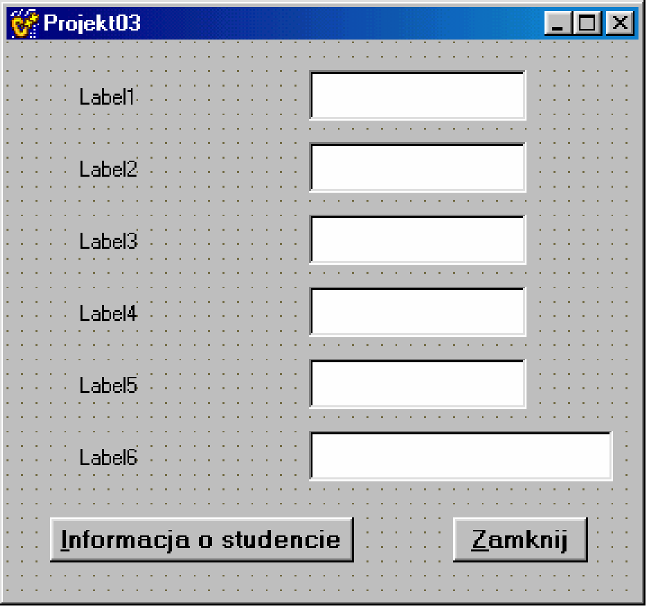
- Plik modułu *Unit02.cpp*. Zawiera kod źródłowy modułu.
- *Unit02.h* zawiera omawianą już definicję klasy formularza.
- *Unit02.dfm* jest plikiem zawierającym definicję obiektu formularza oraz definicje obiektów wszystkich używanych komponentów.

Wykorzystujemy własną strukturę

Ponieważ wiemy już, co to jest formularz i jak jest zorganizowany projekt, bez przeszkód możemy korzystając z C++Buildera uruchomić program, którego kod źródłowy został przedstawiony na wydruku 3.9. Zaprojektujmy w tym celu formularz składający się z 6 komponentów `TEdit`, 6

TLabel oraz dwóch TButton. Sposób rozmieszczenia wymienionych komponentów pokazany jest na rysunku 4.8. Cechy Text komponentów TEdit wyczyśćmy, cechy Caption przycisków Button1 oraz Button2 zmieńmy odpowiednio na &Informacja o studencie oraz &Zamknij.

Rys. 4.8. Sposób rozmieszczenia komponentów na formularzu aplikacji *Projekt03.bpr*



Cechy Caption komponentów TLabel zmienimy w funkcji FormCreate(). Aby dostać się do jej wnętrza wystarczy dwa razy kliknąć obszar formularza. Wydruk 4.4 przedstawia kod modułu *Unit03.cpp* aplikacji *Projekt03.bpr* wykorzystującej napisaną przez nas wcześniej (wydruk 3.9) deklarację ogólnego typu struktury Student. Funkcja obsługi zdarzenia Button1Click() uruchamia zdarzenie polegające na wyświetleniu aktualnej informacji o danej osobie. Informacja ta przechowywana jest w statycznej strukturze Student1 będącej oczywiście typu Student.

Usunięto: w

Usunięto: e

Wydruk 4.4. Kod źródłowy modułu *Unit03.cpp* aplikacji wykorzystującej definicję struktury Student.

```
#include <vcl.h>
#pragma hdrstop
#include "Unit03.h"
#pragma package(smart_init)
#pragma resource "*.dfm"

TForm1 *Form1;

struct Student       // deklaracja ogólnego typu struktury Student
  {
   char  Imie[20];
   char  Nazwisko[20];
   float EgzaminMatematyka;
   float EgzaminFizyka;
   float EgzaminInformatyka;
   char  JakiStudent[30];
  };

//----------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//----------------------------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
```

```
{
   Label1->Caption = "Imię";
   Label2->Caption = "Nazwisko";
   Label3->Caption = "Ocena z fizyki";
   Label4->Caption = "Ocena z matematyki";
   Label5->Caption = "Ocena z informatyki";
   Label6->Caption = "Opinia";
}
//---------------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{

  static struct Student
  Student1 = {"Wacek", "Jankowski", 5, 5, 5, "bardzo dobry student"};

  Edit1->Text = Student1.Imie;
  Edit2->Text = Student1.Nazwisko;
  Edit3->Text = Student1.EgzaminFizyka;
  Edit4->Text = Student1.EgzaminMatematyka;
  Edit5->Text = Student1.EgzaminInformatyka;
  Edit6->Text = Student1.JakiStudent;
}
//---------------------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{
    Application->Terminate();
}
//---------------------------------------------------------------------
```

Ćwiczenie do samodzielnego wykonania

Ćwiczenie 4.2

Posługując się algorytmem pokazanym na wydruku 4.4 uzupełnij go samodzielnie o możliwość wyświetlenia informacji o drugim studencie. Możemy to zrobić w funkcji obsługi odrębnego zdarzenia lub wykorzystując instrukcję `if(...)` w tej samej funkcji.

Usunięto: 1.

Usunięto: w

Wykorzystujemy własną funkcję

Zapoznamy się teraz z jedną z metod umieszczania w programie pisanym w C++Builderze własnej funkcji. W tym celu wykorzystamy, skonstruowaną przez nas wcześniej, funkcję obliczającą kolejne potęgi liczby 2 (zob. wydruk 3.8). Formularz projektu naszej aplikacji, nazwijmy ją *Projekt04.bpr*, składać się będzie z dwóch przycisków `Button1` oraz `Button2` reprezentujących klasę `TButton`. Wykorzystamy też komponent edycyjny `TMemo`.

Samodzielnie napisaną funkcję możemy umieszczać w kodzie źródłowym aplikacji na parę sposobów. Oto kilka z nich.

Usunięto: a

Usunięto: Do najczęściej stosowanych należą:

1. Definicję funkcji umieszczamy w sposób najprostszy z możliwych:

```
TForm1 *Form1;
int power(int x, int y) // definicja funkcji power
{
   int z = 1, i;
   for(i = 1; i <= y; i++)
       z = z * x;
   return z;
}
```

Wywołanie funkcji następuje w kontekście obsługi danego zdarzenia i nie różni się niczym od jej wywołania stosowanego w „tradycyjnym” C++.

Usunięto: api

2. Drugi sposób jest tylko trochę bardziej skomplikowany, mianowicie funkcje definiujemy korzystając z konwencji `__fastcall`:

```
TForm1 *Form1;
int __fastcall power(int x, int y) // definicja funkcji power
{
 ...
}
```

Zastosowanie tej konwencji spowoduje, że trzy pierwsze parametry funkcji mogą zostać przekazane przez rejestry procesora. Mimo takiej modyfikacji wywołanie funkcji pozostaje dalej „tradycyjne”.

Usunięto: ¶
Usunięto: Użycie
Usunięto: zapewni nam

3. Istnieje też możliwość, aby nasza funkcja stała się jawnym obiektem klasy formularza `TForm1`. Należy wówczas jej definicję nagłówkową uzupełnić o nazwę klasy, do której ma przynależeć wraz z operatorem rozróżnienia zakresu:

```
int __fastcall TForm1::power(int x, int y)
{
 ...
}
```

Jednak w tym przypadku należy umieścić jej definicję również w definicji klasy znajdującej się w pliku z rozszerzeniem *.h* w jednej z sekcji, np.:

```
class TForm1 : public TForm
{
__published:      // IDE-managed Components
        TButton *Button1;
        TButton *Button2;
        TMemo *Memo1;
        void __fastcall Button1Click(TObject *Sender);
        void __fastcall Button2Click(TObject *Sender);
        void __fastcall FormCreate(TObject *Sender);
private:   // User declarations
        int __fastcall power(int x, int y); // własna funkcja power()
public:           // User declarations
        __fastcall TForm1(TComponent* Owner);
};
```

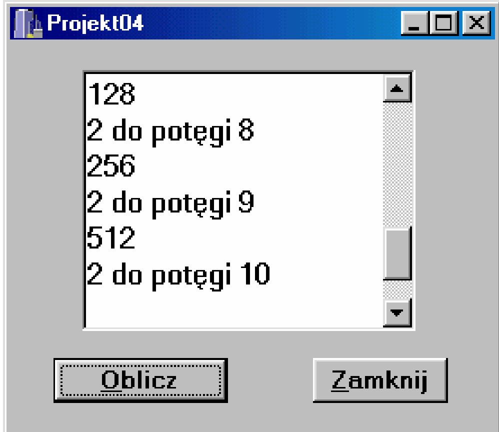
W tym wypadku klasa potrzebuje zdefiniowania prototypu funkcji.

Korzystając ze sposobu włączenia własnej funkcji do definicji klasy zyskujemy bardzo wiele, mianowicie wewnątrz naszej funkcji możemy bez problemów odwoływać się do innych obiektów formularza. Wszystkie własności, cechy, zdarzenia i metody właściwe tym obiektom będą widoczne w naszej funkcji.

Usunięto: w ciele
Usunięto: ,
Usunięto: w

Na rysunku 4.9 pokazano wygląd działającej aplikacji obliczającej kolejne potęgi liczby 2. Znana nam funkcja `power()`, realizująca to zagadnienie, została zdefiniowana jako element klasy formularza, przez co wewnątrz niej można umieścić komponent, w tym wypadku `Memo1`, w którym wyświetlamy odpowiednie informacje dotyczące wykonywania aktualnego potęgowania. Kompletny kod zastosowanego przeze mnie algorytmu został zamieszczony na wydruku 4.5.

Usunięto: w jej ciele



Rys. 4.9. Aplikacja obliczająca kolejne całkowite potęgi liczby 2

Wydruk 4.5. Moduł *Unit04.cpp* aplikacji *Projekt04.bpr* wykorzystującej definicję funkcji `power()`.

```
#include <vcl.h>
#pragma hdrstop
#include "Unit04.h"
#pragma package(smart_init)
#pragma resource "*.dfm"

TForm1 *Form1;

int __fastcall TForm1::power(int x, int y) // definicja funkcji power
{
   Memo1->Lines->Add("2 do potęgi "+IntToStr(y));
   int z = 1, i;
   for(i = 1; i <= y; i++)
       z = z * x;
   return z;
}
//--------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//--------------------------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  Memo1->ScrollBars = ssVertical;
}
//--------------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
for( int i = 1; i <= 10; i++)
     Memo1->Lines->Add(power(2,i));

}
//--------------------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{
    Application->Terminate();
}
//--------------------------------------------------------------------
```

Ćwiczenie do samodzielnego wykonania

Ćwiczenie 4.3.

Usunięto: 2.

1. Posługując się algorytmem pokazanym na wydruku 4.5 przetestuj zaprezentowane sposoby umieszczania własnej funkcji w aplikacji pisanej w C++Builderze.
2. W ten sam sposób przetestuj działanie funkcji, w której parametry deklarowane są przy pomocy wskaźników.

Usunięto: P

Podsumowanie

W niniejszym rozdziale zostały przedstawione niezbędne wiadomości na temat teorii organizacji projektu (aplikacji) pisanego w środowisku Borland C++Builder 5 wykorzystującego elementy programowania zorientowanego obiektowo. Elementy teorii organizacji projektu zostały uzupełnione o konkretne przykładowe rozwiązania prowadzące do zrozumienia ogólnych zasad tworzenia aplikacji. Zapoznaliśmy się również z paroma praktycznymi sposobami wykorzystania inspektora obiektów. Wyjaśnione zostały pojęcia klasy, konstruktora klasy oraz funkcji obsługi zdarzenia. Samodzielnie wykonana prosta aplikacja pozwoli też zrozumieć, co to jest zdarzenie i w jaki sposób, z poziomu funkcji obsługujących wybrane zdarzenia, odwoływać się do innych obiektów aplikacji. Zostało też pokazane, w jaki sposób możemy odwoływać się do samodzielnie napisanych struktur oraz funkcji i jak uczynić je równoprawnymi obiektami formularza.

Rozdział 5 . Podstawowe elementy biblioteki VCL

Na potrzeby tej książki komponentami nazywać będziemy te obiekty, które możemy pobrać z palety komponentów i umieścić je na formularzu aplikacji. Czynność te przećwiczyliśmy w poprzednim rozdziale. Komponenty VLC są podstawowymi elementami, z których budujemy aplikację. W rozdziale tym omówimy krótko podstawowe komponenty VCL oraz hierarchię ich ważności.

Usunięto: ą

Hierarchia komponentów VCL

W ogólnym przypadku rozróżniamy cztery podstawowe rodzaje komponentów:

- Komponenty standardowe. Są one najczęściej używane przez programistów, dlatego większość z nich umieszczona jest na pierwszej karcie palety komponentów — karcie Standard.
- Komponenty sterujące. Nie są one dostępne w bibliotece standardowej.
- Komponenty graficzne. Służą do wypisywania tekstu bezpośrednio na formularzu oraz do wyświetlania grafiki.
- Komponenty niewidoczne. Stają się niewidoczne po uruchomieniu programu. Wszystkie komponenty z karty Dialogs oraz niektóre z kart System i Servers są obiektami, które przestajemy widzieć w działającej aplikacji.

Usunięto: –

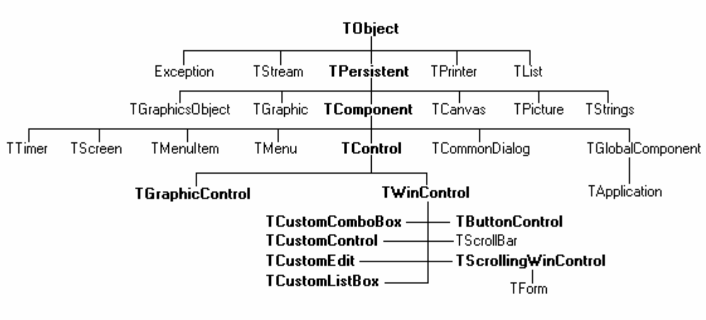
Usunięto: e

Na rysunku 5.1 pokazano fragment hierarchii obiektów biblioteki VCL. Tutaj przedstawiony został jedynie fragment drzewa obiektów Borland C++ Buildera 5, ale najlepszym sposobem zapoznania się z całością zagadnienia jest obejrzenie dosyć obszernego arkusza przedstawiającego wszystkie obiekty. Arkusz taki dostajemy zawsze wraz z licencją na kompilator.

Usunięto: P

Usunięto: gdyż

Rys. 5.1. Dziedziczenie klas biblioteki VCL



Łatwo możemy zauważyć, że główną część drzewa powyższych obiektów stanowi sześć klas.

Klasa TObject

Jest przodkiem wszystkich typów obiektowych Borland C++ Builder 5. Najczęściej nie korzysta się bezpośrednio z właściwości i metod, które nam udostępnia.

Klasa TPersistent

Wszystkie typy obiektowe, mające zdolność posługiwania się strumieniami, pochodzą właśnie od tej klasy. Klasa ta w rzeczywistości nie definiuje nowych właściwości ani pól, definiuje natomiast destruktor `~TPersistent()` oraz sześć metod:

`Assign()` – metoda przypisania obiektowi właściwości i atrybutów innego obiektu.

`AssignTo()` – metoda odwrotna do poprzedniej. Przypisuje danemu obiektowi kopię własnych właściwości i atrybutów.

`DefineProperties()` – ta metoda definiuje sposób przypisania strumieniowi pewnych dodatkowych właściwości komponentu.

`GetNamePath()` – umożliwia odczytanie nazwy obiektu oraz jego ustalonych właściwości w inspektorze obiektów.

`GetOwner()` – podaje właściciela obiektu.

`TPersistent()` – tworzy nowy obiekt.

Z dokładniejszym opisem oraz praktycznymi sposobami wykorzystania tych metod możemy się zapoznać sięgając do plików pomocy.

Klasa TComponent

Z klasy tej pochodzi każdy komponent C++ Buildera 5. Wprowadzone przez nią właściwości i metody pozwalają na obsługę komponentów poprzez inspektora obiektów. Z niektórymi z nich zapoznaliśmy się przy okazji tworzenia ostatniego projektu.

Klasa TControl

Komponenty wizualne reprezentowane w tej klasie są widoczne w czasie działania programu, chociaż istnieją sposoby by je ukryć lub uczynić niewidocznymi w trakcie działania programu. Obiekty tej klasy posiadają szereg właściwości (z niektórymi z nich zapoznaliśmy się już wcześniej). Oto niektóre z nich.

Właściwości klasy TControl

`Align` — określa, w jaki sposób komponent ma być ustawiony na formularzu (obszarze klienta). Jeżeli np. wybierzemy w inspektorze obiektów `alClient`, wówczas komponent ten pokryje cały dostępny obszar formularza. Właściwość tego typu aktywna jest np. dla komponentów typu `TPanel`, `TGroupBox` czy `TRadioGroup` z karty Standard.

`Anchors` — określa położenie komponentu w stosunku do jednego z rogów formularza.

`Caption` — opisuje komponent. Ćwiczyliśmy to już na przykładzie tytułu formularza, czy chociażby opisu przycisków `TButton`.

`ClientHeight` oraz `ClientWidth` — określa wymiary komponentu (wysokość i długość) w obszarze klienta.

`Color` — ustala kolor wypełnienia (wnętrza) komponentu.

`Cursor` — określa postać kursora, który będzie widoczny w obszarze danego komponentu.

`DragKind` oraz `DragMode` — działanie ich było pokazywane już w tej książce.

`Enabled` — określa, czy komponent będzie dla nas dostępny. Jeżeli posługując się np. przyciskiem typu `TButton` napiszemy:

```
Button1->Enabled = FALSE;
```

przycisk będzie widoczny, ale nie będzie aktywny. Powrót do normalnej sytuacji możliwy jest dzięki:

```
Button1->Enabled = TRUE;
```

Analogicznych ustawień dokonamy też przy pomocy inspektora obiektów.

`Font` — ustala rodzaj czcionki napisów widocznych w obszarze komponentu.

`Hint` — ta właściwość sprawia, że można wpisać „dymek podpowiedzi", ale wówczas `ShowHint` musi być ustalone jako `TRUE`.

`Height` i `Width` — określają rozmiar komponentu.

`Text` — dzięki tej właściwości tekst wyświetlany jest na obszarze komponentu. Stosujemy ją m.in. do obiektów typu `TEdit`.

`Top`, `Left` — określają odległości komponentu od krawędzi odpowiednio górnej i lewej formularza (lub ogólnie innego komponentu, od którego wywodzi się komponent, któremu cechy te przypisujemy).

`Visible` — określa, czy komponent ma być widoczny. Jeżeli w programie napiszemy:

```
Button1->Visible = FALSE;
```

komponent pozostanie całkowicie niewidoczny do czasu wywołania:

```
Button1->Visible = TRUE;
```

Czynność tę można również wykonać przy pomocy inspektora obiektów.

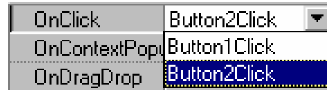
Zdarzenia klasy TControl

Klasa `TControl` udostępnia nam również szereg pożytecznych zdarzeń. Do najczęściej używanych należą:

Usunięto: –
Usunięto: my
Usunięto: –
Usunięto: –
Usunięto: u
Usunięto: –
Usunięto: –
Usunięto: my
Usunięto: –
Usunięto: wybieramy
Komentarz: Ale na czym polegało?
Usunięto: –
Usunięto: –
Usunięto: my
Usunięto: P
Usunięto: –
Usunięto: my
Usunięto: –
Usunięto: ujemy
Usunięto: –
Usunięto: –
Usunięto: w
Usunięto: tą
Usunięto: właściwość m. in.
Usunięto: –
Komentarz: ?
Usunięto: –
Usunięto: ci
Usunięto: ą
Usunięto: też

`OnClick` — po kliknięciu obszaru komponentu zostanie wywołana funkcja obsługi wybranego zdarzenia. Można wyobrazić sobie sytuację, gdy mamy np. dwa przyciski typu `TButton` i z każdym z nich skojarzona jest funkcja odpowiedniego zdarzenia (takie operacje już nie są dla nas tajemnicą). Powiedzmy, że chcemy szybko zamienić role tych przycisków, tzn. aby kliknięcie `Button1` wywoływało funkcję obsługi zdarzenia `Button2Click()`, wówczas zaznaczając `Button1`, w inspektorze obiektów zamieniamy je po prostu rolami, tak jak pokazuje to rysunek 5.2.

Rys. 5.2. Przypisanie przyciskowi `Button1` funkcji obsługi zdarzenia skojarzonego z `Button2`



`OnDblClick` — dwukrotne kliknięcie obszaru komponentu spowoduje wywołanie funkcji odpowiedniego zdarzenia.

`OnResize` — wywołuje np. funkcję obsługi zdarzenia po zmianie rozmiaru komponentu.

`OnMouseDown` — wywołuje reakcję na zdarzenie polegające na kliknięciu komponentu.

`OnMouseMove` — każdy ruch myszką nad komponentem wywoła funkcję odpowiedniego zdarzenia.

`OnMouseUp` — jak wyżej, tyle że w przypadku puszczenia przycisku muszki.

`TControl` udostępnia nam również zdarzenia związane z przesuwaniem komponentów przy pomocy myszki: `OnDragOver`, `OnDragDrop`, `OnEndDrag`, `OnStartDock` czy `OnStartDrag`. Jako przykład użycia tych pożytecznych zdarzeń niech nam posłuży poniższy wydruk.

Wydruk 5.1. Idea posługiwania się zdarzeniami `OnMouseMove` oraz `OnStartDock`. W przykładzie tym ustawienia właściwości przycisku `Button1` muszą być podobne do tych z rysunku 4.6.

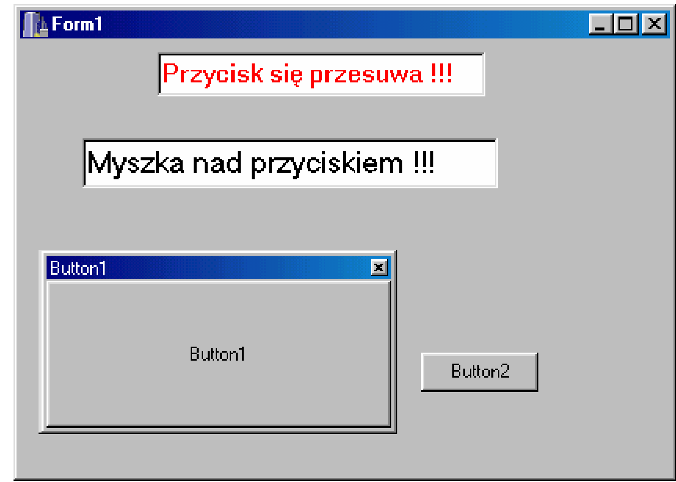
```
#include <vcl.h>
#pragma hdrstop
#include "Unit1.h"
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//--------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//-------------------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{
   Application->Terminate();
}
//--------------------------------------------------------------------

void __fastcall TForm1::Button1StartDock(TObject *Sender,
                                        TDragDockObject *&DragObject)
{
    Canvas->TextOut(50, 50, "Przycisk się przesuwa !!!");
}
//--------------------------------------------------------------------
void __fastcall TForm1::Button1MouseMove(TObject *Sender,
                                      TShiftState Shift, int X, int Y)
{
    Edit1->Top = Y;
    Edit1->Left = X;
    Edit1->Text = "Myszka nad przyciskiem!!!"
```

Usunięto: –
Usunięto: w
Usunięto: na
Usunięto: –
Usunięto: w
Usunięto: ze
Usunięto: –
Usunięto: anie
Usunięto: i
Usunięto: –
Usunięto: a
Usunięto: e
Usunięto: nad
Usunięto: em.
Usunięto: –
Usunięto: –

```
}
//--------------------------------------------------------------------
```

Formularz działającej aplikacji powinien przedstawiać się tak, jak pokazuje to rys. 5.3.

Usunięto: a



Rys. 5.3. Formularz działającej aplikacji wykorzystującej zdarzenia `OnMouseMove` oraz `OnStartDock` udostępniane przez klasę `TControl`

Klasa TGraphicControl

Reprezentuje nieaktywne komponenty wykorzystywane w różnego rodzaju operacjach związanych z grafiką. Widoczne na ekranie, komponenty te mogą wyświetlać tekst lub grafikę. Z najpowszechniej stosowanych komponentów tego typu należy wymienić: `TBevel`, `TImage`, `TPaintBox`, `TShape`, `TSpeedButton`, `TSplitter` oraz `TCustomLabel`, od którego wywodzą się z kolei `TDBText` i `TLabel`. Komponenty tego rodzaju mogą nie tylko obsługiwać zdarzenia, których źródłem jest myszka, ale również mogą być używane w funkcjach obsługi innych zdarzeń. Jako przykład praktycznego wykorzystania jednego z komponentów `TLabel` niech nam posłuży przykład funkcji obsługi zdarzenia reagującego na zmianę położenia myszki na formularzu:

Usunięto: Będąc
Usunięto: w
Usunięto: ymi
Usunięto: ,
Usunięto: oraz
Usunięto: typu
Usunięto: jak
Usunięto: e
Usunięto: takich

```
void __fastcall TForm1::OnMoseMove(TObject *Sender, TShiftState Shift,
                                   int X, int Y)
{
   Label1->Font->Style = TFontStyles() << fsBold;
   Label1->Font->Size = 16;
   Label1->Font->Color = clBlue;
   Label1->Top = Y;
   Label1->Left = X;
   Label1->Caption = "Tekst ciagniety za myszką X=" + IntToStr(X) +
                     +" Y= " +IntToStr(Y);
}
```

W wyniku, na obszarze klienta, zobaczymy odpowiedni napis oraz aktualne współrzędne kursora myszki. Funkcja ta została zbudowana w bardzo prosty sposób. Na obszar formularza przeniosłem komponent typu `TLabel`. Następnie, raz klikając formę, w karcie zdarzeń inspektora obiektów wybrałem `OnMouseMove`, któremu przypisałem identyczną nazwę. Dalej, przy pomocy klawisza Enter mogłem dostać się już do wnętrza odpowiedniej funkcji. Wartości numeryczne współrzędnych zostały zamienione na tekst przy pomocy wielce użytecznej funkcji:

Usunięto: W
Usunięto: na
Komentarz: Co? Zdarzenie?
Usunięto: Następnie
Usunięto: żna
Komentarz: Wydaje mi się, że to słowo trzeba jakoś wyróżnić, ale nie wiem jak.

```
extern PACKAGE AnsiString __fastcall IntToStr(int Value);
```

Funkcja ta konwertuje dane typu int na dane typu AnsiString. Zauważmy też, że wykorzystaliśmy tutaj właściwość Caption określającą łańcuch znaków wyświetlanych na komponencie.

Komentarz: J.w.
Usunięto: Konwertującej
Komentarz: J.w.

Klasa TWinControl

Wszystkie okna edycyjne, listy wyboru, przyciski itp. są obiektami potomnymi tej klasy. Komponenty okienkowe mogą być aktywne, posiadają swoje własne identyfikatory oraz możliwość przewijania. Klasa ta posiada szereg właściwości, metod i zdarzeń. Wykorzystanie kilku z nich prezentuje poniższy przykład:

Usunięto: Są

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  Memo1->Brush->Color = clBlue;
  Memo1->Font->Color = clYellow;
  Memo1->Font->Style = TFontStyles() << fsBold;
  Memo1->Left = ClientRect.Left;
  Memo1->Text = "Tekst";
  Form1->ScrollBy(1,1);
  Memo1->ScrollBy(1,1);
}
```

Metody klasy TWinControl

W wyniku cyklicznego (cyklicznych kliknięć myszką na przycisk `Button1`) wywoływania funkcji obsługi zdarzenia `Button1Click()` zauważymy, że w oknie edycji `Memo1` pojawi się pewien napis, ponadto będzie on przewijany w tym oknie. Również cały obszar klienta formularza będzie się przemieszczał. Jest to możliwe dzięki metodzie `ScrollBy()`, która przesuwa całe okno dodając do jego aktualnych współrzędnych wartości argumentów, z którymi metodą tą wywołaliśmy. Z innych ciekawych metod należy wymienić `CanFocus()` — sprawdzającą, czy dany komponent okienkowy może być uaktywniony, `Focused()` — sprawdzającą czy okienko jest aktywne i wreszcie `SetFocus()` — uaktywniającą wybrany komponent okienkowy. Aktywny komponent powinien reagować na zdarzenia (np. kliknięcie jego obszaru). Jeżeli nie chcemy zbyt często używać myszki, zawsze można przenieść aktywność z jednego okienka do drugiego w zupełnie inny sposób. Na przykład, pracując w okienku `Memo1` zapragniemy nagle, by okno `Memo2` stało się aktywne, wówczas w odpowiednim miejscu kodu wystarczy wpisać:

Usunięto: Dokonamy tego korzystając z metody

Komentarz: Przy pomocy których wywołaliśmy tę metodę?

Usunięto: w

Usunięto: ze

```
Memo2->SetFocus();
```

Lub, gdy jakąś operację uzależniamy od tego, czy dane okienko jest aktywne, możemy użyć prostej konstrukcji:

```
void __fastcall TForm1::Memo2Change(TObject *Sender)
{
   if (Memo2->Focused() == TRUE)
      Memo1->Color = clGreen;
}
```

Właściwości klasy TWinControl

Przykładem właściwości udostępnianych przez `TWinControl` i wykorzystanych już przez nas będą `Brush` — ustalający kolor wypełnienia, a także `ClientRect` i `Left` przesuwające komponent do prawego rogu formularza.

Usunięto: oraz

Komentarz: Left do prawego rogu?

Zdarzenia klasy TWinControl

Zdarzenia udostępniane przez wymienioną klasę w stosunku do komponentów okienkowych obsługiwane są z poziomu klawiatury oraz przy pomocy myszki. Możliwe jest również przenoszenie aktywności pomiędzy okienkami. Do najważniejszych zdarzeń generowanych z poziomu klawiatury należą: `OnKeyPress`, `OnKeyDown`, `OnKeyUp` oraz `OnEnter` i `OnExit`.

Usunięto: ,

Usunięto: oraz

Usunięto: m

Jako przykład ich wykorzystania pokażmy jak można zmieniać kolor obszaru komponentu okienkowego, np. TMemo:

```
//--------------------------------------------------------------------
void __fastcall TForm1::OnEnterMemo1(TObject *Sender)
{
   Memo1->Color = clBlue;
}
//--------------------------------------------------------------------
void __fastcall TForm1::OnExitMemo2(TObject *Sender)
{
   Memo1->Color = clRed;
}
//--------------------------------------------------------------------
void __fastcall TForm1::OnEnterMemo2(TObject *Sender)
{
    Memo2->Color = clYellow;
    Memo1->Color = clLime;
}
//--------------------------------------------------------------------
```

Oczywiście, należy pamiętać, że zdarzenie OnExit działa najlepiej w odpowiedzi na naciśnięcie tabulatora (Tab).

Podsumowanie

W rozdziale tym dokonano przeglądu podstawowych elementów biblioteki VCL. Krótko zostały omówione najważniejsze klasy dostępne w tej bibliotece. Parę pożytecznych przykładów, opisujących sposoby praktycznego wykorzystania metod właściwości i zdarzeń udostępnianych przez poszczególne klasy, pomoże nam zrozumieć ideę korzystania z biblioteki VCL.

Usunięto: krótkiego

Rozdział 6
Biblioteka VCL

Najważniejszym elementem środowisk programistycznych dla Windows, takich jak Delphi czy Builder, jest biblioteka wizualnych komponentów. W ogólności, korzystając z Borland C++Buildera 5 możemy posługiwać się dziewiętnastoma paletami takich komponentów:

- Standard components
- Additional components
- Win32 components
- System components
- Data Access components
- Data Controls components
- ADO components
- InterBase components
- MIDAS components
- InternetExpress components
- Internet components
- FastNet components
- Decision Cube components
- QReport components
- Dialogs components
- Win 3.1 components
- Samples components
- ActiveX components
- Servers components

W wersji Standard mamy do dyspozycji dziesięć kart zawierających najczęściej używane komponenty. Nie jest oczywiście możliwe, aby w opracowaniu o niewielkich rozmiarach szczegółowo opisać każdy komponent z uwzględnieniem jego cech, metod i zdarzeń, nawet jeżeli pracujemy w standardowej wersji C++Buildera 5. Ważnym uzupełnieniem muszą być dla nas pliki pomocy Buildera. Sprowadzając jakiś komponent do obszaru formularza zawsze możemy posłużyć się klawiszem F1, aby otrzymać naprawdę wyczerpującą informację na temat klasy, do jakiej należy wybrany komponent, jego właściwości, itp. Poruszając się po niezwykle bogatych w treści plikach pomocy, przy odrobinie wysiłku znajdziemy tam również bardzo wiele pożytecznych przykładów praktycznego posługiwania się określonymi obiektami. Obecnie zapoznamy się z kilkoma najczęściej używanymi kartami.

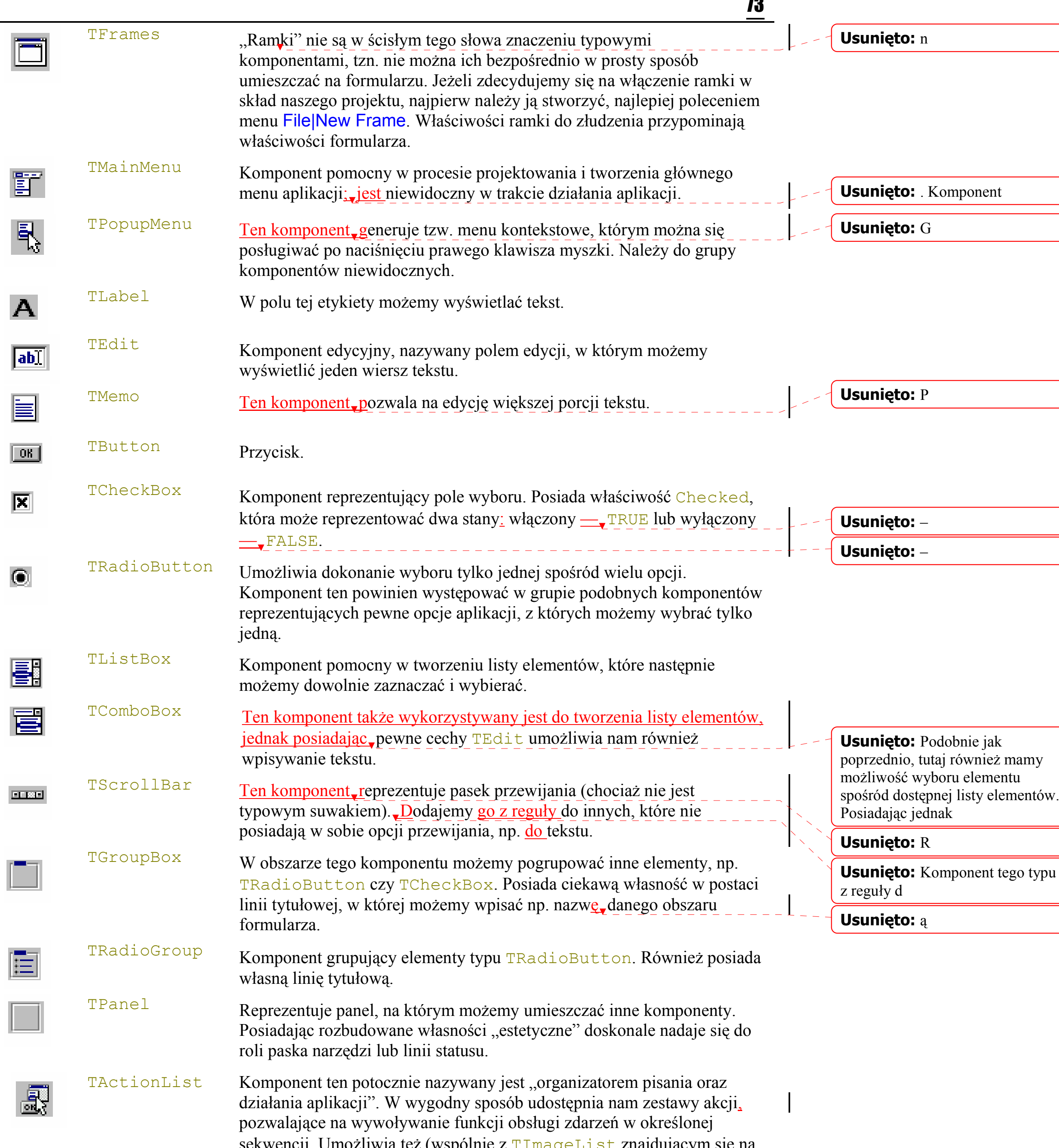
Karta Standard

Korzystając z zasobów tej karty mamy do dyspozycji wszystkie najczęściej wykorzystywane komponenty reprezentujące sobą wszystkie podstawowe elementy sterujące Windows.

Komentarz: Czy chodzi o: „[...] komponenty, które są jednocześnie podstawowymi elementami sterującymi sysytemem Windows”?

Tabela 6.1. *Komponenty karty Standard*

Ikona	Typ	Znaczenie



Komponent	Opis
TFrames	„Ramki” nie są w ścisłym tego słowa znaczeniu typowymi komponentami, tzn. nie można ich bezpośrednio w prosty sposób umieszczać na formularzu. Jeżeli zdecydujemy się na włączenie ramki w skład naszego projektu, najpierw należy ją stworzyć, najlepiej poleceniem menu File|New Frame. Właściwości ramki do złudzenia przypominają właściwości formularza.
TMainMenu	Komponent pomocny w procesie projektowania i tworzenia głównego menu aplikacji; jest niewidoczny w trakcie działania aplikacji.
TPopupMenu	Ten komponent generuje tzw. menu kontekstowe, którym można się posługiwać po naciśnięciu prawego klawisza myszki. Należy do grupy komponentów niewidocznych.
TLabel	W polu tej etykiety możemy wyświetlać tekst.
TEdit	Komponent edycyjny, nazywany polem edycji, w którym możemy wyświetlić jeden wiersz tekstu.
TMemo	Ten komponent pozwala na edycję większej porcji tekstu.
TButton	Przycisk.
TCheckBox	Komponent reprezentujący pole wyboru. Posiada właściwość Checked, która może reprezentować dwa stany: włączony — TRUE lub wyłączony — FALSE.
TRadioButton	Umożliwia dokonanie wyboru tylko jednej spośród wielu opcji. Komponent ten powinien występować w grupie podobnych komponentów reprezentujących pewne opcje aplikacji, z których możemy wybrać tylko jedną.
TListBox	Komponent pomocny w tworzeniu listy elementów, które następnie możemy dowolnie zaznaczać i wybierać.
TComboBox	Ten komponent także wykorzystywany jest do tworzenia listy elementów, jednak posiadając pewne cechy TEdit umożliwia nam również wpisywanie tekstu.
TScrollBar	Ten komponent reprezentuje pasek przewijania (chociaż nie jest typowym suwakiem). Dodajemy go z reguły do innych, które nie posiadają w sobie opcji przewijania, np. do tekstu.
TGroupBox	W obszarze tego komponentu możemy pogrupować inne elementy, np. TRadioButton czy TCheckBox. Posiada ciekawą własność w postaci linii tytułowej, w której możemy wpisać np. nazwę danego obszaru formularza.
TRadioGroup	Komponent grupujący elementy typu TRadioButton. Również posiada własną linię tytułową.
TPanel	Reprezentuje panel, na którym możemy umieszczać inne komponenty. Posiadając rozbudowane własności „estetyczne” doskonale nadaje się do roli paska narzędzi lub linii statusu.
TActionList	Komponent ten potocznie nazywany jest „organizatorem pisania oraz działania aplikacji”. W wygodny sposób udostępnia nam zestawy akcji, pozwalające na wywoływanie funkcji obsługi zdarzeń w określonej sekwencji. Umożliwia też (wspólnie z TImageList znajdującym się na karcie Win32) bardzo estetyczne zaprojektowanie menu aplikacji.

Usunięto: n

Usunięto: . Komponent

Usunięto: G

Usunięto: P

Usunięto: –

Usunięto: –

Usunięto: Podobnie jak poprzednio, tutaj również mamy możliwość wyboru elementu spośród dostępnej listy elementów. Posiadając jednak

Usunięto: R

Usunięto: Komponent tego typu z reguły d

Usunięto: ą

TFrames

Przy okazji omawiania komponentów karty Standard wspominaliśmy, że sposób wykorzystania elementów tej klasy jest nieco odmienny do pozostałych obiektów oferowanych przez bieżącą kartę. Z tego względu należy poświęcić mu odrębny fragment rozdziału. Jedynym sposobem zapoznania się z podstawowymi regułami stosowania `TFrames` jest zaprojektowanie odpowiedniej aplikacji. Tradycyjnie już stwórzmy na dysku nowy katalog, powiedzmy, że nazwiemy go *\Projekt05*. Zaprojektujemy naszą aplikację w ten sposób, że będzie składać się z głównego formularza, dwóch przycisków klasy `TButton` oraz jednego komponentu `TFrames`. Zbudujemy naprawdę prostą aplikację, której jedynym celem będzie wyświetlenie wyniku powstałego z dodania dwóch dowolnych liczb. Część aplikacji realizującej to zadanie przypiszemy obiektowi `TFrames`. Zacznijmy więc od zaprojektowania reprezentanta `TFrames` w naszym formularzu.

Zastosowanie TFrames

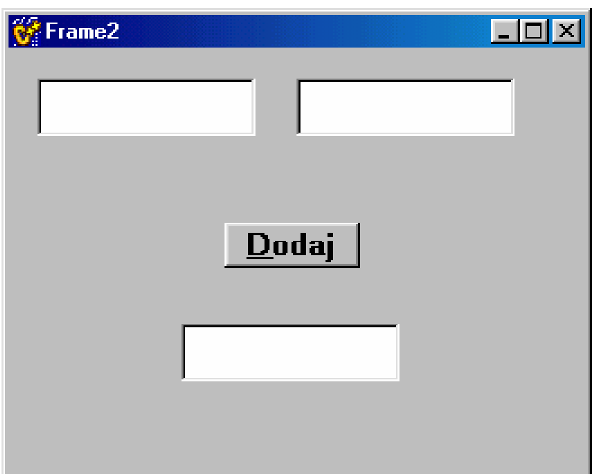
Ćwiczenie 6.1.

1. Poleceniem menu File|New Frame stwórzmy obiekt ramki. Jego cesze `Name` powinna być automatycznie przypisana nazwa `Frame2`, ponadto powinien być on widoczny w obszarze głównego formularza. Zauważmy, że `Frame2` posiada większość cech głównego formularza łącznie z obszarem klienta. Już teraz możemy ustalić niektóre jej cechy, np. `DragKind` oraz `DragMode` tak jak pokazuje to rys. 4.6.

Usunięto: ,

Usunięto: j

3. W obszarze ramki rozmieśćmy trzy komponenty klasy `TEdit` oraz jeden `TButton`. Cechy `Text` komponentów reprezentowanych przez `Edit1`, `Edit2` oraz `Edit3` wyczyśćmy. Cechę `Name` przycisku `Button1` zmieńmy na &Dodaj. Rozmieszczenie poszczególnych elementów w obszarze obiektu `TFrames` reprezentowanego przez `Frame2` i potocznie nazywanego ramką powinno być podobne jak na rys. 6.1.



Rys. 6.1. Przykładowe rozmieszczenie komponentów w obszarze ramki `Frame2`

4. Teraz dodajmy funkcję obsługi zdarzenia wywoływanego w odpowiedzi na naciśnięcie przycisku zatytułowanego Dodaj. Dwukrotnie klikając jego obszar bez problemu dostajemy się do wnętrza funkcji `Button1Click()`. Wypełnimy ją następującym kodem:

Usunięto: Dodajemy

Usunięto: w

```
void __fastcall TFrame2::Button1Click(TObject *Sender)
{
  try
    {
     Edit3->Text = FloatToStr(StrToFloat(Edit1->Text) +
```

```
                                          StrToFloat(Edit2->Text));
                }
                catch(...)
                {
                  ShowMessage("Błąd !. Nieodpowiedni format danych.");
                }
```

Funkcje `StrToFloat()` dokonają konwersji ciągu znaków przechowywanych w cechach `Text` komponentów `Edit1` oraz `Edit2` na zmiennopozycyjną postać numeryczną, czyli po prostu na liczby z przecinkami. Używając prostego operatora „`+`" te dwie wartości dodamy do siebie, zaś wynik działania zostanie z kolei przypisany cesze `Text` komponentu edycyjnego reprezentowanego przez `Edit3`. Aby postać numeryczna liczby mogła być wyświetlona w oknie edycyjnym musi zostać zamieniona na łańcuch znaków (najlepiej typu `AnsiString`). Dokonamy tego stosując funkcję `FloatToStr()` konwertującą postać numeryczną liczby zmiennopozycyjnej na odpowiedni łańcuch znaków. Całość operacji dodawania została ujęta w klauzule `try...catch(...)` (por. wydruk 4.3). Powiemy, że zastosowaliśmy prostą obsługę wyjątków, którą w pewnym uproszczeniu można przedstawić jako ciąg instrukcji:

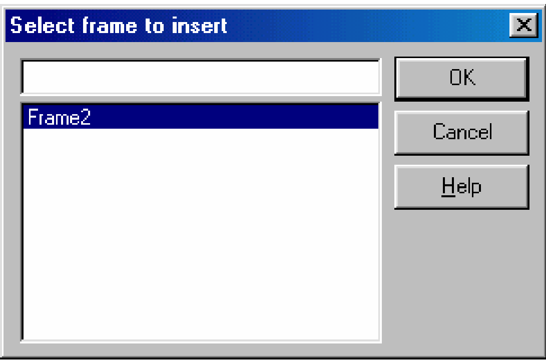
Usunięto: a

```
            try // „próbuj" wykonać operację
                {
                 // ciag wykonywanych operacji
                }
                catch(...)  // jeżeli operacja nie powidła się „przechwyć wyjątek"
                {
                  // przetwarzanie wyjątku
                  // jeżeli nastąpił wyjątek pokaż komunikat
                }
```

W naszym prostym przykładzie wyjątkiem będzie albo nie wpisanie liczb(y) w ogóle do komponentów edycyjnych, albo wpisanie znaku nie będącego liczbą. Chociaż jest to bardzo prosty przykład korzystania z wyjątków, jednak powinniśmy starać się zrozumieć ich naturę. Wyjątki są również obiektami Windows pełniąc tam bardzo ważną rolę. Jeżeli takowy wyjątek wystąpi, odpowiedni komunikat pokażemy w postaci okienka z przyciskiem korzystając z funkcji `ShowMessage()`.

W taki oto sposób przygotowaliśmy obiekt ramki. Należy obecnie włączyć go do naszego formularza. To, że ramka jest wyświetlana na formularzu jeszcze nic nie znaczy, musimy ją jawnie dołączyć do projektu. Dokonamy tego właśnie dzięki komponentowi `Frames` z karty Standard. W tym celu kliknijmy na obszar głównego formularza, ramka powinna się „schować", następnie wybierzmy z karty opcję `Frames` i ponownie kliknijmy na formularzu. Wynik naszego działania powinien przybrać postać pokazaną na rys. 6.2.

Rys. 6.2. Włączanie w skład głównego formularza obiektu typu `TFrames`



Potwierdzając przyciskiem OK. uzyskamy pożądany efekt. `Frame2` od tej pory stanie się obiektem składowym naszej aplikacji. Już teraz możemy obejrzeć dyrektywy prekompilatora w module głównego projektu. Pojawiła się tam dyrektywa `#pragma link "Unit2"`, co oznacza, że konsolidator dołączy ten plik do głównego projektu. Domyślnie wszystkie moduły skojarzone z `Frame2` będą posiadały nazwy *Unit2.**, zaś te przyporządkowane głównemu projektowi — *Unit1.**. Jeżeli już na tym etapie zdecydujemy się zapisać nasz projekt na dysku, np. pod nazwą

Komentarz: Czy to prawidłowe określenie?

Usunięto: e

Projekt05.bpr, na pewno zauważymy, że zostanie utworzony tylko jeden projekt projekt głównego formularza, zaś ramka stanie się po prostu jego częścią.

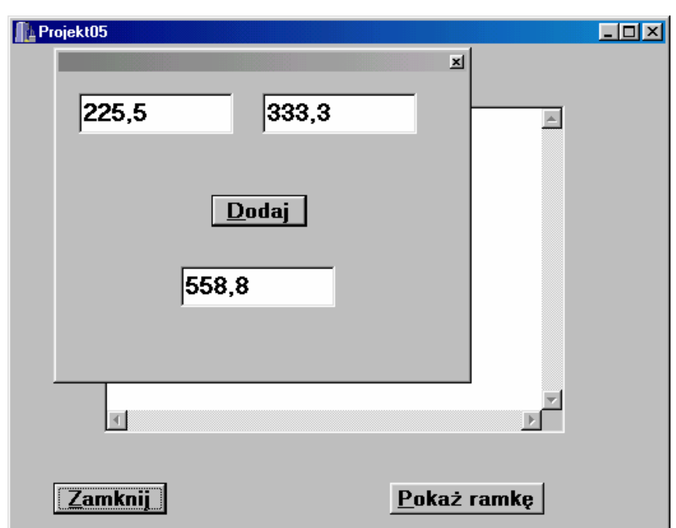
Usunięto: –

Usunięto: ła

Pozostaje nam już teraz zaprojektować dwa proste zdarzenia reprezentujące zamknięcie aplikacji oraz ewentualnie ponowne wyświetlenie ramki (przydatne, gdy ramkę zamkniemy korzystając z jej własnego pola zamknięcia). W tym celu umieśćmy na formularzu dwa komponenty typu `TButton` i przypiszmy im funkcje obsługi odpowiednich zdarzeń. Kod źródłowy modułu naszego projektu pokazany jest na wydruku 6.1, zaś rysunek 6.3 przedstawia działającą aplikację.

Wydruk 6.1. Moduł *Unit1.cpp* aplikacji *Projekt05.dpr*.

```
#include <vcl.h>
#pragma hdrstop
#include "Unit1.h"
#pragma package(smart_init)
#pragma link "Unit2"
#pragma resource "*.dfm"
TForm1 *Form1;
//-------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//---------zamyka całą aplikację-------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
   Application->Terminate();
}
//----------uaktywnia Frame2-----------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{
   Frame21->Show();
}
//-------------------------------------------------------------------
```



Rys. 6.3. Komponent `TFrames` jako integralna część aplikacji

Obsługa programu sprowadza się do wprowadzenia z klawiatury dwóch liczb i wykonania ich dodawania. Część dziesiętną wpisujemy po przecinku (tak, jak w Kalkulatorze Windows). Zauważmy, że komponent reprezentowany przez `Frame2` możemy swobodnie przesuwać po całym ekranie. Można go zamknąć, a następnie z powrotem otworzyć korzystając z przycisku Pokaż ramkę, z którym skojarzona jest funkcja obsługi zdarzenia `Button2Click()`. Z pojedynczym formularzem możemy skojarzyć bardzo wiele ramek, z których każda może pełnić

Usunięto: u

Usunięto: ,

Usunięto: m

rolę odrębnej aplikacji, wykonując właściwe jej zadania. Tego typu obiekty mogą być również osadzane na stałe w obszarze formularza.

Wykorzystanie pozostałych komponentów karty Standard

Aby zilustrować właściwości niektórych pozostałych komponentów stworzymy przykładową aplikację w postaci dosyć złożonego okienka. Głównym zadaniem aplikacji będzie odczytanie z dysku przykładowego pliku tekstowego oraz odpowiednie wyświetlenie jego zawartości w komponencie edycyjnym TMemo. Będziemy mieli ponadto możliwość zmiany koloru tła dla wczytywanego tekstu oraz koloru i kroju czcionki.

Usunięto: jej

Formularz naszej aplikacji, nazwijmy ją *Projekt06.bpr* , składać się będzie z pojedynczych pól edycji TEdit i TMemo. W ich reprezentantach, Edit1 oraz Memo1 będziemy odpowiednio wpisywali nazwę pliku do odczytu oraz wyświetlali jego zawartość. Aby zawartość pliku ładnie się wyświetlała, cechę WordWrap obiektu Memo1 ustalmy jako TRUE, wówczas tekst będzie się zwijał w okienku. Ponadto, w przypadku odczytu większego pliku, dobrze by było mieć do dyspozycji możliwość jego przewijania w okienku. Uzyskamy to ustalając dla cechy ScrollBars opcję ssBoth — zawartość okna będzie przewijana zarówno w pionie jak i poziomie. W obszarze określonym komponentem klasy TGroupBox i reprezentowanym przez GroupBox1, w którym zmieniać będziemy styl i kolor czcionki umieśćmy dwa obiekty klasy TCheckBox. Każdemu z nich przypiszemy funkcje obsługi zdarzeń:

Usunięto: ę

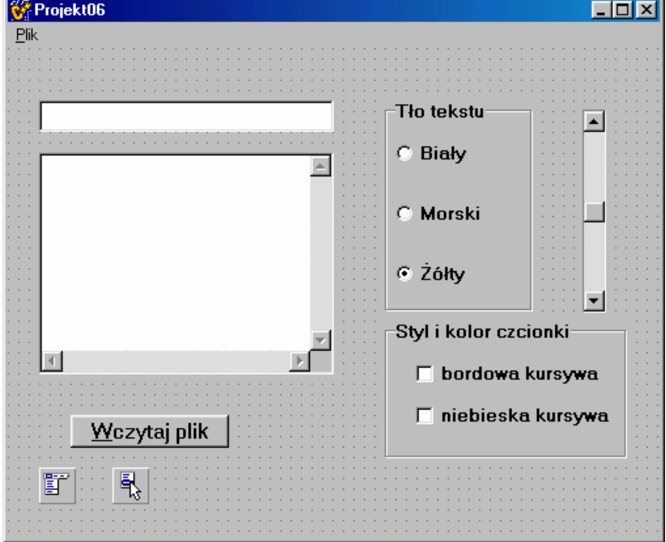
Usunięto: na

Usunięto: –

```
void __fastcall TForm1::CheckBox1Click(TObject *Sender)
{
   if (CheckBox1->State == TRUE)
     {
      Memo1->Font->Name = fsItalic;
      Memo1->Font->Color = clMaroon;
     }
}
//--------------------------------------------------------------------
void __fastcall TForm1::CheckBox2Click(TObject *Sender)
{
   if (CheckBox2->Checked)
     {
      Memo1->Font->Name = fsItalic;
      Memo1->Font->Color = clBlue;
     }
}
```

których wykonanie spowoduje, że będziemy mogli zmienić krój czcionki oraz jej kolor w tekście wyświetlanym w Memo1. Dokonaliśmy pierwszych przypisań w naszym projekcie. Zawsze dobrym zwyczajem jest sprawdzenie ich poprawności. Aby tego dokonać musimy niestety w jakiś sposób wczytać wybrany plik (lub wpisać coś z klawiatury). Nie przejmujmy się, że tym razem zrobimy to nieco „na piechotę”. C++Builder posiada oczywiście odpowiednie narzędzia pozwalające na pełną automatyzację podobnych czynności, niemniej jednak przypomnienie sobie paru istotnych pojęć związanych z plikami na pewno nikomu z nas nie zaszkodzi.

Usunięto: ,



Rys. 6.4. Rozmieszczenie elementów edycyjnych i sterujących na formularzu projektu *Projekt06.bpr*

Wczytujemy plik z dysku

Plik, którego pełną nazwę wraz z rozszerzeniem będziemy w tym konkretnym przypadku wpisywać ręcznie w okienku `Edit1`, wczytamy posługując się jednym ze sposobów właściwych dla C++, tzn. najpierw plik otworzymy do odczytu korzystając z funkcji `FileOpen()`. Funkcja ta posiada dwa parametry. Pierwszy z nich mówi, że korzystając z metody `c_str()` zwracającej wskaźnik (`char *`) do pierwszego znaku łańcucha identyfikującego właściwość `Text` obiektu `Edit1`: `Edit1->Text.c_str()`, przechowujemy wskaźnik do nazwy pliku (a dokładniej do pierwszego znaku tej nazwy). Drugi parametr, w naszym wypadku określa, że plik jest otworzony w trybie do odczytu (*file mode Open Read*). Otworzonemu plikowi przypisujemy jego identyfikator `iFileHandle`. Następnie przy pomocy funkcji `FileSeek(iFileHandle, 0, 2)` sprawdzimy, czy plik nie jest pusty, tzn. określimy gdzie jest jego koniec, mówiąc dokładniej wskaźnik pliku umieścimy na jego końcu. Jeżeli mamy kłopoty ze zrozumieniem znaczenia wskaźnika pliku, wyobraźmy sobie, że oglądamy wywołaną kliszę fotograficzną, możemy ją przeglądać klatka po klatce. Wskaźnik pliku jest właśnie taką „klatką”, która umożliwia nam jego przeszukiwanie. Wywołana z sukcesem funkcja ta zwróci nam rozmiar pliku, który będziemy przechowywać pod zmienną `iFileLenght`. W przypadku błędnego wywołania zwróci wartość `-1`. Wówczas należy zadbać, by aplikacja powiadomiła nas o tym przykrym fakcie — dokonamy tego wywołując funkcję `MessageBox()`, generującą wymagany przez nas komunikat okienkowy. Jeżeli wszystko będzie w porządku, znowu wywołamy `FileSeek()`, tylko tym razem ustawimy się na początku pliku. Kolejnym etapem będzie przydzielenie bufora, tzn. obszaru pamięci, w którym będziemy przechowywać zawartość pliku. W tego typu sytuacjach najlepiej jest to zrobić wykorzystując operator `new`, który dynamicznie przydzieli tyle obszaru pamięci, ile potrzeba dla naszych danych, czyli `iFileLength + 1`. Jedynkę musimy dodać, ponieważ posługujemy się ciągiem znaków zakończonych tzw. zerowym ogranicznikiem (zerowym bajtem), który wyraźnie zaznacza koniec danych w pliku. Ponieważ elementy tablic w C++ zaczynają się od zerowego indeksu, więc po to, by nie zgubić ostatniego bajtu, należy do odczytanego rozmiaru pliku dodać jedynkę. Kolejnym etapem będzie przeczytanie zawartości bufora danych, identyfikowanego przez wskaźnik `Buffer` przy pomocy funkcji `FileRead()`. Następnie plik zamykamy funkcją `FileClose()`, a zawartość bufora wczytujemy do okna reprezentowanego przez `Memo1` przy pomocy metody `Append()`. Na koniec, korzystając z operatora `delete`, zwalniamy wykorzystywany obszar pamięci. Wszystkie omówione operacje zostały zebrane na wydruku 6.2, ilustrującym funkcję obsługi zdarzenia generowanego po naciśnięciu przycisku `Button1`, którego cechę `Caption` w naszej aplikacji zamieniliśmy na &Wczytaj plik.

Usunięto: ,
Usunięto: która
Usunięto: W p
Usunięto: m
Usunięto: –
Usunięto: z powodu, że
Usunięto: reprezentującym treść
Usunięto: i

Wydruk 6.2. Funkcja obsługi zdarzenia `Button1Click()` wykorzystywana w projekcie *Projekt06.bpr*

```
// --- wczytanie pliku
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  int iFileHandle;
  int iFileLength;
  char *Buffer;
      iFileHandle = FileOpen(Edit1->Text.c_str(), fmOpenRead);
      iFileLength = FileSeek(iFileHandle, 0, 2);
      if(iFileLength == -1)
        {
         Application->MessageBox("Nie można otworzy pliku.",
                                 "Błąd pliku", IDOK);
        }
         else
           {
            FileSeek(iFileHandle, 0, 0);
            Buffer = new char[iFileLength+1];
            FileRead(iFileHandle, Buffer, iFileLength);
            FileClose(iFileHandle);
            Memo1->Lines->Append(Buffer);
            delete [] Buffer;
           }
}
```

Skoro poruszyliśmy temat operatorów `new` i `delete`, wyjaśnienie jeszcze jednej rzeczy ułatwi nam w przyszłości pracę, już przy samodzielnym pisaniu aplikacji. Wspomnieliśmy wcześniej o idei obsługi wyjątków. Otóż okazuje się, że wymienione operatory, a zwłaszcza `new`, są bardzo mocno osadzone na tej arenie. W naszym przykładzie nie zastosowaliśmy żadnej obsługi wyjątków, jednak w programach bardziej zaawansowanych koniecznym byłoby całość instrukcji, począwszy od miejsca wywołania funkcji `FileOpen()` aż po miejsce, w którym używamy operatora `delete`, ująć w klauzule `try...catch()`. Klauzule te przechwytują pewien wyjątek, a następnie przetwarzają go korzystając chociażby z prostego komunikatu. Powinniśmy pamiętać, że jeżeli nie można przydzielić wystarczającej ilości pamięci do wczytania pliku, operator `new` wygeneruje własny wyjątek. W konsekwencji aplikacja, która tego wyjątku nie przechwyci, może zostać zakończona w sposób niekontrolowany.

Usunięto: Ponieważ książka ta zatytułowana jest „Borland C++Builder 5. Ćwiczenia praktyczne", wydaje się więc, że s

Usunięto: pomoże nam w przyszłości bardzo ułatwić sobie

Komentarz: ?

Usunięto: cych

Usunięto: orzenie

> Posługując się parą operatorów `new` i `delete` zawsze należy pamiętać, że `delete` można używać jedynie ze wskaźnikami do obszarów pamięci, które zostały uprzednio przydzielone przy pomocy operatora `new`. Używając `delete` z innym adresem możemy popaść w poważne kłopoty.

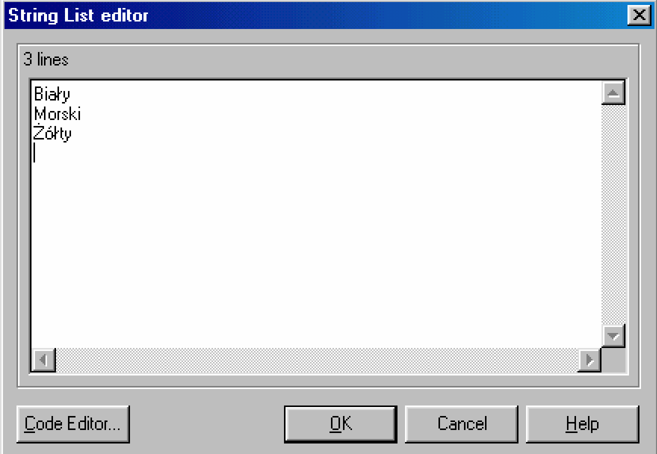
Komponenty TRadioGroup oraz TScrollBar

Sposób wykorzystywania tych komponentów sprawia nam niekiedy pewne trudności. Już sam opis `TRadioGroup` może być nieco mylący, gdyż sugeruje, że wystarczy umieścić go na formularzu, a następnie w jego obszarze ulokować reprezentantów klasy `TradioButton`, pobranych bezpośrednio z palety komponentów. Niestety, takie podejście nic nam nie da. Aby reprezentant klasy `TRadioGroup` spełniał rzeczywiście jakąś rolę w naszej aplikacji, należy odwołać się w inspektorze obiektów do jego cechy `Items`, następnie rozwinąć `TStrings` i w pojawiającym się oknie edycji dokonać odpowiedniego opisu opcji. W naszym przypadku, komponent ten będzie odpowiedzialny za zmianę koloru tła wyświetlanego tekstu, zatem należy tam wpisać np.:

Usunięto: e

Usunięto: jamy

Usunięto: ujemy



Rys. 6.5. String List Editor w akcji

I oczywiście potwierdzić. Następnie w opcję `Columns` (inspektor obiektów) wpiszemy `1`, zaś do `ItemIndex` wstawimy `2` (pamiętajmy, że numeracja opcji zaczyna się od 0). Wystarczy teraz klikając dwa razy dostać się do wnętrza funkcji obsługi zdarzenia `RadioGroup1Click()` i wypełnić ją odpowiednim kodem. Przy okazji od razu możemy włączyć do programu komponent reprezentowany przez `ScrollBar1` z wykorzystaniem jego cechy `Position`, tak jak pokazano poniżej:

```
void __fastcall TForm1::ScrollBar1Change(TObject *Sender)
{
    RadioGroup1->ItemIndex = ScrollBar1->Position;
}
//-------------------------------------------------------------------
void __fastcall TForm1::RadioGroup1Click(TObject *Sender)
{
    if (RadioGroup1->ItemIndex == 0)
      Memo1->Color = clWhite;
    if (RadioGroup1->ItemIndex == 1)
      Memo1->Color = clAqua;
    if (RadioGroup1->ItemIndex == 2)
      Memo1->Color = clYellow;
}
//-------------------------------------------------------------------
```

Poprawność zastosowanych przypisań należy oczywiście (najlepiej od razu) przetestować.

Komponenty TMainMenu oraz TPopupMenu

Zajmijmy się teraz obiektami, przy pomocy których będziemy tworzyć opcje menu zarówno głównego, jak i kontekstowego. Aby dostać się do okna służącego do tworzenia menu głównego, należy zaznaczyć komponent `MainMenu1`, a następnie dwukrotnie kliknąć myszką pole `Items` karty zdarzeń inspektora obiektów (oczywiście, ten sam efekt otrzymamy klikając dwukrotnie samą ikonę na formularzu). Zmieńmy cechę `Caption` (nagłówek) na &Plik, pojawi się wówczas nowe pole obok naszej opcji. W ten sposób możemy tworzyć nawet bardzo rozbudowane menu, ale o tym wszystkim jeszcze sobie powiemy w dalszej części książki. Teraz jednak wskażmy pole poniżej i cesze `Caption` przypiszmy &Wczytaj Plik, następnie przejdźmy do karty Events inspektora obiektów i zdarzeniu `OnClick` przypiszmy `Button1Click`. Klikając teraz dwa razy pole &Wczytaj Plik od razu znajdziemy się wewnątrz procedury obsługi zdarzenia `Button1Click()`. Powróćmy do okna Form1->MainMenu1 i przejdźmy niżej, następną opcję zatytułujemy &Zamknij aplikację. W karcie zdarzeń inspektora obiektów jej cechę `Name` zmieńmy na `ApplicationClose`, zaś w karcie zdarzeń zdarzeniu `OnClick` przypiszmy `ApplicationCloseClick`. Dwukrotnie klikając dostaniemy się do wnętrza funkcji obsługi zdarzenia `ApplicationCloseClick()`, którą wypełnimy odpowiednim, znanym już kodem.

Usunięto: w

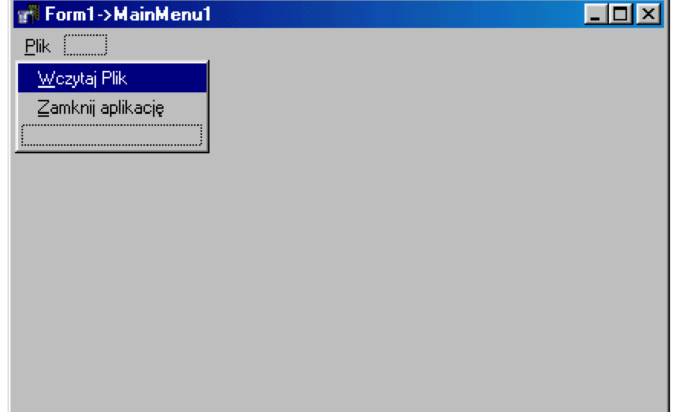
Usunięto: na

Usunięto: e

Usunięto: chodzimy

Usunięto: ujemy

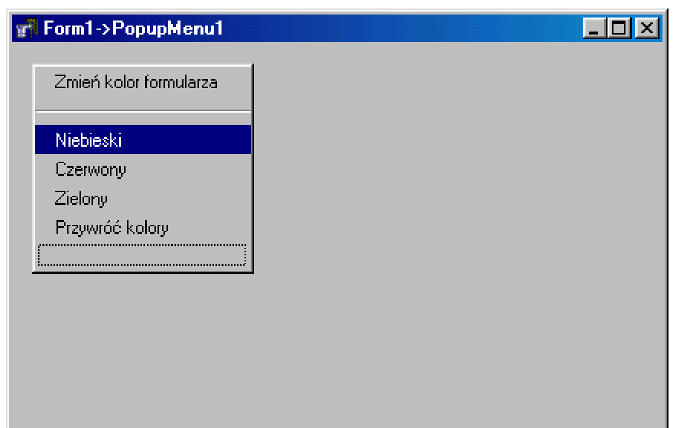
Usunięto: w polu



Rys. 6.6. Okno służące do projektowania głównego menu wraz ze zdefiniowanymi, przykładowymi opcjami

Sposób tworzenia menu kontekstowego nie różni się w istocie od tego, co już powiedzieliśmy na temat menu głównego. Aby wyglądało ono tak, jak pokazuje rysunek 6.7, należy cechy `Caption` poszczególnych jego opcji zmienić odpowiednio na Niebieski, Czerwony, Zielony i Przywróć kolory, zaś cechy `Name` odpowiednio na Blue, Red, Green oraz BtnFace. Teraz w karcie Events każdemu z nich wystarczy przypisać zdarzenia `BlueClick`, `RedClick`, `GreenClick` oraz `BtnFaceClick`. Klikając dwukrotnie poszczególne opcje menu dostaniemy się do funkcji obsługi odpowiednich zdarzeń, których wnętrza wypełnimy kodem pokazanym na wydruku 6.3. Należy też pamiętać, że po to, by menu kontekstowe poruszało się za myszką, należy skorzystać z metody `Popup()`, uaktywniającej menu, wywoływanej w funkcji obsługi zdarzenia `Form1MouseDown()`, właściwej dla całego formularza. Innymi słowy, aby się tam dostać, należy raz kliknąć formularz, przejść do karty zdarzeń i wybrać zdarzenie `OnMouseDown`. Żeby dostać się do zdarzenia `FormCreate`, należy dwa razy kliknąć sam formularz.

Usunięto: na
Usunięto: , tzn.
Usunięto: na
Usunięto: u
Usunięto: na
Usunięto: ym
Usunięto: u.



Rys. 6.7. Zdefiniowane, przykładowe opcje menu kontekstowego TPopupMenu

TPanel oraz TCheckBox

Pokazany na rysunku 6.4 sposób umieszczenia na formularzu reprezentantów klas `TPanel` oraz `TCheckBox` nie powinien stanowić dla nas problemu. Pamiętajmy, że obiekt `Panel1` nie będzie generował żadnych zdarzeń, służy on tylko do grupowania innych obiektów. Chcąc dostać się do wnętrza funkcji obsługi zdarzeń przyporządkowanych odpowiednio komponentom `CheckBox1` oraz `CheckBox2` należy po prostu dwukrotnie je kliknąć. Zdarzenia generowane przez te komponenty posłużą nam do zmiany koloru i kroju czcionki tekstu wyświetlanego w `Memo1`.

Usunięto: tylko,
Usunięto: na nie

Przykładowa aplikacja

Wydruk 6.3 przedstawia kompletny kod głównego modułu naszej aplikacji. Obsługa programu sprowadza się do wpisania z klawiatury w oknie edycji `Edit1` nazwy pliku wraz z rozszerzeniem, który chcemy obejrzeć. Pamiętać jednak należy, iż w naszym algorytmie nie zastosowaliśmy żadnych opcji umożliwiających programowi rozróżnianie wielkości wpisywanych liter. Dlatego nazwę wczytywanego pliku należy wpisać z ewentualnym uwzględnieniem małych i dużych liter.

Wydruk 6.3. Kompletny kod modułu *Unit06.cpp* projektu *Projekt06.bpr*

```
#include <vcl.h>
#include <stdio.h>
#pragma hdrstop
#include "Unit06.h"
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//--------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//--------------------------------------------------------------------
void __fastcall TForm1::ScrollBar1Change(TObject *Sender)
{
    RadioGroup1->ItemIndex = ScrollBar1->Position;
}
//--------------------------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{
     PopupMenu1->AutoPopup = FALSE;
}
//--------------------------------------------------------------------
void __fastcall TForm1::Form1MouseDown(TObject *Sender,
      TMouseButton Button, TShiftState Shift, int X, int Y)
{
     PopupMenu1->Popup(X, Y);
}
//--------------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  int iFileHandle;
  int iFileLength;
  char *Buffer;
      iFileHandle = FileOpen(Edit1->Text.c_str(), fmOpenRead);
      iFileLength = FileSeek(iFileHandle, 0, 2);
      if(iFileLength == -1)
        {
         Application->MessageBox("Nie można otworzy pliku.",
                                 "Błąd pliku", IDOK);
        }
         else
           {
            FileSeek(iFileHandle, 0, 0);
            Buffer = new char[iFileLength+1];
            FileRead(iFileHandle, Buffer, iFileLength);
            FileClose(iFileHandle);
            Memo1->Lines->Append(Buffer);
            delete [] Buffer;
           }
}
//--------------------------------------------------------------------
void __fastcall TForm1::ApplicationCloseClick(TObject *Sender)
{
    switch(MessageBox(NULL, " Działanie aplikacji zostanie"
                      " zakończone.", "Uwaga!",
                       MB_YESNOCANCEL | MB_ICONQUESTION))
        {
         case ID_YES     : Application->Terminate();
         case ID_CANCEL : Abort();
        }
```

```
}
//---------------------------------------------------------------------
void __fastcall TForm1::CheckBox1Click(TObject *Sender)
{
   if (CheckBox1->State == TRUE)
     {
      Memo1->Font->Name = fsItalic;
      Memo1->Font->Color = clMaroon;
     }
}
//---------------------------------------------------------------------
void __fastcall TForm1::CheckBox2Click(TObject *Sender)
{
   if (CheckBox2->Checked)
     {
      Memo1->Font->Name = fsItalic;
      Memo1->Font->Color = clBlue;
     }
}
//---------------------------------------------------------------------
void __fastcall TForm1::RadioButton1Click(TObject *Sender)
{
     Memo1->Color = clYellow;
}
//--------------------------------------------------------------------
void __fastcall TForm1::RadioButton2Click(TObject *Sender)
{
     Memo1->Color = clAqua;
}
//---------------------------------------------------------------------
void __fastcall TForm1::RadioButton3Click(TObject *Sender)
{
    Memo1->Color = clWhite;
}
//---------------------------------------------------------------------
void __fastcall TForm1::RadioGroup1Click(TObject *Sender)
{
    if (RadioGroup1->ItemIndex == 0)
      Memo1->Color = clWhite;
    if (RadioGroup1->ItemIndex == 1)
      Memo1->Color = clAqua;
    if (RadioGroup1->ItemIndex == 2)
      Memo1->Color = clYellow;
}
//---------------------------------------------------------------------
void __fastcall TForm1::BlueClick(TObject *Sender)
{
    Form1->Color = clBlue;
}
//---------------------------------------------------------------------
void __fastcall TForm1::RedClick(TObject *Sender)
{
     Form1->Color = clRed;
}
//---------------------------------------------------------------------
void __fastcall TForm1::GreenClick(TObject *Sender)
{
     Form1->Color = clGreen;
}
//---------------------------------------------------------------------
void __fastcall TForm1::BtnFaceClick(TObject *Sender)
{
    Form1->Color = clBtnFace;
}
//---------------------------------------------------------------------
```

Ćwiczenie do samodzielnego wykonania

Ćwiczenie 6.2.

Postaraj się umieścić na formularzu obok menu Plik inne menu, np. Kolory. Zaprojektuj zdarzenia tego menu w ten sposób, by można było dzięki nim zmieniać tło obiektu Mamo1.

Hierarchia własności obiektów Właściciele i rodzice

Każdy obiekt przez nas wykorzystywany posiada dwie podstawowe własności: może być właścicielem (ang. *owner*) innych obiektów lub może być ich rodzicem (ang. *parent*). Istnieje subtelna różnica pomiędzy tymi dwoma pojęciami, z której należy zdawać sobie sprawę, jeżeli naprawdę zechcemy zrozumieć idee rządzące zasadami programowania obiektowo -zdarzeniowego.

Wykorzystując graficzny interfejs użytkownika GUI (ang. *Graphics User Interface*) budujemy aplikacje, których głównym elementem jest formularz. Formularz jest właścicielem obiektów, które na nim umieszczamy. Jako przykład rozpatrzmy komponent CheckBox1, znajdujący się na obszarze określonym przez GroupBox1. Jeżeli zechcemy teraz dowolnie zmienić położenie CheckBox1, napotkamy pewne trudności, mianowicie nie będziemy mogli przesunąć go poza GroupBox1. Mówimy, że GroupBox1, czyli reprezentant klasy TGroupBox stał się rodzicem dla CheckBox1, reprezentującego klasę TCheckBox. Aby przeanalizować przykład hierarchii własności, sprawdźmy kto jest właścicielem formularza. W tym celu wystarczy zaprojektować nowe zdarzenie (lub wykorzystać istniejące) i napisać:

```
ShowMessage(Form1->Owner->ClassName());
```

Przekonamy się, że właścicielem formularza jest aplikacja. Jeżeli natomiast chcielibyśmy sprawdzić w ten sposób, czy formularz ma rodzica, wygenerujemy po prostu wyjątek. Formularz w prostej linii nie posiada rodzica.

Następnie napiszmy:

```
ShowMessage(GroupBox1->Owner->ClassName());
ShowMessage(GroupBox1->Parent->ClassName());
```

Pojawiający się komunikat nie pozostawia cienia wątpliwości: zarówno właścicielem, jak i rodzicem komponentu GroupBox1, umieszczonego bezpośrednio na formularzu, jest TForm1.

Przechodząc dalej sprawdzimy cechy własności komponentu CheckBox1:

```
ShowMessage(CheckBox1->Parent->ClassName());
```

Stwierdzimy, że jego rodzicem jest TGroupBox, zaś właścicielem:

```
ShowMessage(CheckBox1->Owner->ClassName());
```

Pozostanie dalej formularz, czyli TForm1.

Zdarzają się sytuacje, kiedy potrzebujemy, nawet w trakcie działania aplikacji, zmienić położenie jakiegoś komponentu umieszczonego uprzednio w obszarze takim jak TGroupBox czy TPanel. Aby to zrobić, wystarczy pamiętać o omówionych relacjach własności. Jeżeli chcemy, by np. CheckBox1 znalazł się bezpośrednio w innym miejscu formularza, wystarczy przypisać mu Form1 jako rodzica, a następnie podać nowe współrzędne:

```
CheckBox1->Parent = Form1;
CheckBox1->Top = 20;
```

Usunięto: .
Usunięto: O
Usunięto: P
Usunięto: –
Usunięto: w

```
CheckBox1->Left = 50;
```

> Należy rozróżniać pojęcia właściciela (`Owner`) i rodzica (`Parent`). Rodzic nie jest tożsamy z właścicielem. Właściciela określa się tylko raz podczas wywoływania jego konstruktora i nie można już go zmienić bez zniszczenia obiektu. Rodzica obiektu możemy natomiast zmienić zawsze.

Przedstawione powyżej wiadomości na temat własności i rodzicielstwa obiektów stanowią tylko wierzchołek góry lodowej. Być może dla niektórych z nas wyda się to nieco zaskakujące, ale mamy też możliwość samodzielnego tworzenia i umieszczania na formularzu różnych obiektów (dostępnych oczywiście w bibliotece VCL). Musimy wiedzieć, że każdy taki obiekt posiada swojego własnego konstruktora, jednak opis tego zagadnienia nie mieści się w ramach tej książki.

Usunięto: właścicielstwa

Ćwiczenie do samodzielnego wykonania

Ćwiczenie 6.3.

Umieść na formularzu komponent typu `TPanel`, na nim zaś jeszcze parę innych komponentów widzialnych (mogą być umieszczone na zasadzie piramidki). Postaraj się samodzielnie określić relacje właścicielstwa i rodzicielstwa pomiędzy nimi.

Karta Additional

Karta Additional jest rozszerzeniem karty Standard. Zawiera szereg komponentów, które okazują się bardzo przydatne w projektowaniu aplikacji.

Tabela 6.2. *Komponenty karty Additional*

Ikona	Typ	Znaczenie
	`TBitBtn`	Przycisk, na którym można umieszczać rysunek.
	`TSpeedButton`	Przycisk umieszczany zwykle na pasku zadań. Na nim również możemy umieszczać rysunki.
	`TMaskEdit`	Komponent służący do maskowania i filtrowania danych wpisywanych zwykle z klawiatury.
	`TStringGrid`	Element, który pozwala na umieszczenie na formularzu typowego arkusza składającego się z komórek edycyjnych rozmieszczonych w wierszach i kolumnach.
	`TDrawGrid`	Element, który umożliwia graficzne przedstawienie danych nie będących tekstem.
	`TImage`	Komponent graficzny. Umożliwia wyświetlenie na formularzu np. mapy bitowej.
	`TShape`	Ten element umieszcza na formularzu wybraną figurę geometryczną. Posiada cechę `Shape`, przy pomocy której możemy wybrać rodzaj figury.
	`TBevel`	Składnik, który umieszcza na formularzu obszar prostokątny, posiadający cechy trójwymiarowości. Dzięki cechom `Shape` i `Style` możemy określić sposób jego wyświetlania.

Usunięto: P

Usunięto: U

Usunięto: U

Usunięto: U

	TScrollBox	Komponent zawierający paski przewijania. Może pełnić rolę przewijanego okienka.
	TCheckListBox	Element stanowiący połączenie listy i pola wyboru. Posiada cechę `Items` umożliwiającą edytowanie tekstu.
	TSplitter	Ten komponent dzieli formularz lub okno na kilka części, których obszar możemy zmieniać. Dopiero użycie co najmniej dwóch takich komponentów może dać pożądany efekt.
	TStaticText	Składnik który jest odpowiednikiem `TLabel`, uzupełniony jednak o szereg właściwości, umożliwia bardziej estetyczne wyświetlenie tekstu.
	TControlBar	Komponent, który umożliwia eleganckie i wygodne rozmieszczenie różnych komponentów na pasku zadań.
	TApplicationEvents	Komponent umożliwiający przechwytywanie zdarzeń generowanych przez aplikację, w tym również wyjątków.
	TChart	Ten składnik umożliwia graficzną wizualizację danych w postaci różnego rodzaju wykresów.

Usunięto: P
Usunięto: D
Usunięto: J
Usunięto: U
Usunięto: U

Karta Win32

Karta zawiera wszystkie elementy sterujące reprezentowane w aplikacjach Windows.

Tabela 6.3. *Komponenty karty Win32*

Ikona	Typ	Znaczenie
	TTabControl	Korzystając z tego komponentu mamy możliwość tworzenia zakładek.
	TPageControl	Komponent składający się z większej ilości kart. Aby stworzyć nową kartę w najprostszym przypadku należy nacisnąć prawy klawisz myszki i wybrać opcję New Page.
	TImageList	Ten składnik umożliwia utworzenie listy elementów graficznych. Każdemu z obrazków automatycznie jest przypisywany odpowiedni indeks. Komponent niewidzialny.
	TTrackBar	Suwak. Posiada cechę `Position`, dzięki której można regulować i odczytywać aktualną pozycję wskaźnika przesuwania.
	TProgressBar	Komponent będący wskaźnikiem postępu. Również posiada cechę `Position`, dzięki której możemy śledzić postęp wykonywanych operacji.
	TUpDown	Komponent umożliwiający zwiększanie bądź zmniejszanie jakiejś wartości. Z reguły nie występuje samodzielnie. Wartości należy wyświetlać w komponentach edycyjnych. Również posiada cechę `Position`.
	THotKey	Element umożliwiający utworzenie klawisza szybkiego dostępu.

Usunięto: U
Usunięto: U

	TAnimate	Komponent, który umożliwia wyświetlanie sekwencji obrazów.
	TDateTimePicker	Komponent będący w istocie pewnego rodzaju kalendarzem. Umożliwia odczytanie i wybranie odpowiedniej daty. Posiada rozwijany obszar podobny do TListBox.
	TMonthCalendar	Komponent bardzo podobny do poprzedniego, z tą różnicą, że wyświetla od razu datę bieżącego miesiąca.
	TTreeView	Składnik powodujący hierarchiczne wyświetlanie elementów.
	TListView	Lista widoków wyświetla pozycje składające się z ikon i etykiet.
	THeaderControl	Komponent tworzący listę nagłówkową mogącą składać się z wielu sekcji.
	TStatusBar	Linia statusu formularza. Aby umieścić odpowiedni tekst w linii statusu formularza, wystarczy nacisnąć prawy klawisz myszki i dostać się do Panels Editor...
	TToolBar	Komponent, który tworzy paski narzędzi.
	TCoolBar	Komponent będący pewną odmiana panelu, z tą różnicą, że pozwala na zmianę jego rozmiaru.
	TPageScroller	Ten składnik może zawierać inne obiekty z możliwością przewijania ich zarówno w pionie jak i poziomie.

Usunięto: U

Usunięto: y

Usunięto: H

Usunięto: Umieszczając go na formularzu

Usunięto: aby

Usunięto: umożliwiającego umieszczenie odpowiedniego tekstu w linii statusu formularza.

Usunięto: T

Usunięto: M

Karta System

Karta System zawiera szereg komponentów wykorzystywanych w różnych operacjach na poziomie systemu Windows.

Tabela 6.4. *Komponenty karty System*

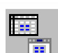
Ikona	Typ	Znaczenie
	TTimer	Jest komponentem niewidzialnym. Służy do generowania zdarzeń w równych odstępach czasu.
	TPaintBox	Komponent wykorzystywany do wykonywania różnych operacji graficznych.
	TMediaPlayer	Komponent, który umożliwia wykorzystywanie w aplikacji technik multimedialnych.
	TOleContainer	Jest komponentem niewidocznym. Służy do generowania na formularzu obszaru klienta OLE.
	TDDEClientConv	Komponent niewidzialny. Umożliwia połączenie z serwerem DDE.
	TDDEClientItem	Komponent niewidzialny. Określa dane wysyłane przez klienta podczas konwersacji DDE.
	TDDEServerConv	Niewidzialny komponent umożliwiający nawiązanie dialogu z klientem DDE.

Usunięto: U

Usunięto: y

Usunięto: Komponent niewidzialny.

Usunięto: Komponent niewidzialny.

Ikona	Typ	Znaczenie
	TDDEServerItem	Komponent niewidzialny. Umożliwia określenie danych wysyłanych do klienta w trakcie konwersacji DDE.

Usunięto: Komponent niewidzialny.

Karta Dialogs

Komponenty Karty Dialogs reprezentują standardowe okna dialogowe Windows. Będą to np. okna do zapisu pliku, odczytu, drukowania, wyboru rodzaju czcionki czy palety kolorów. Wszystkie są komponentami niewidzialnymi.

Usunięto: ,

Tabela 6.5. *Komponenty karty Dialogs*

Ikona	Typ	Znaczenie
	TOpenDialog	Komponent tworzący okienko dialogowe służące do wyboru i otwarcia pliku.
	TSaveDialog	Komponent tworzący okienko dialogowe służące do zapisu danych do pliku.
	TOpenPictureDialog	Składnik umożliwiający dokonanie wyboru plików, w tym również plików graficznych.
	TSavePictureDialog	Komponent tworzący okienko dialogowe służące do zapisu pliku graficznego.
	TFontDialog	Komponent, który umożliwia dokonanie wyboru czcionki.
	TColorDialog	Okienko dialogowe służące do wyboru palety kolorów.
	TPrintDialog	Okienko dialogowe służące do drukowania.
	TPrinterSetupDialog	Komponent określający ustawienia drukarki.
	TFindDialog	Komponent służący do podglądu i wyszukiwania tekstu.
	TReplaceDialog	Okienko, które umożliwia wyszukanie fragmentu tekstu i zastąpienie go innym.

Usunięto: U

Usunięto: U

Usunięto: D

Usunięto: y

Usunięto: U

Usunięto: U

Jako pożyteczny przykład wykorzystania niektórych komponentów z omówionych już kart palety komponentów VCL, wykonamy prostą aplikację wczytującą wybrany plik z dysku i wyświetlającą jego zawartość w obiekcie edycyjnym `TRichEdit`. Postaramy się jednak przy tej okazji zaprojektować naprawdę „profesjonalne" menu.

Tworzymy profesjonalne menu

Ćwiczenie 6.4.

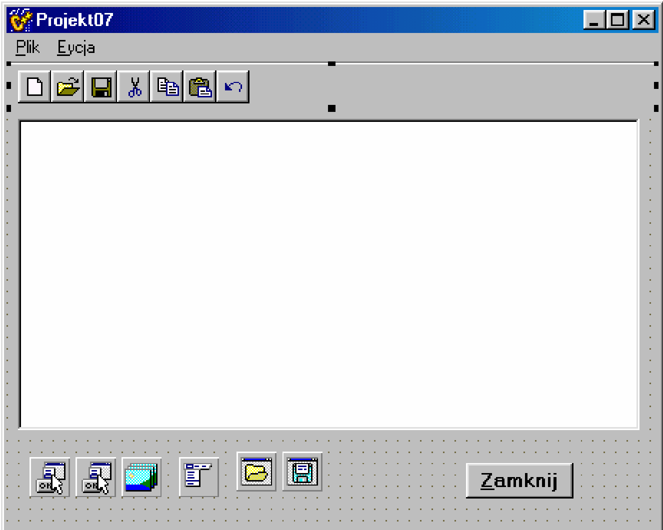
1. Załóżmy na dysku oddzielny katalog, powiedzmy, że nazwiemy go *\Projekt07*.
2. Jeżeli w katalogu, w którym zainstalowaliśmy Buildera istnieje podkatalog *\Buttons*, odszukajmy go i wybierzmy 7 map bitowych pokazanych poniżej. Przekopiujmy je do naszego katalogu. Jeżeli zaś nie jesteśmy w stanie ich odszukać, należy skorzystać z edytora graficznego pokazanego na rys. 1.25. Postarajmy się samodzielnie wykonać podobne obrazki (zapisując je oczywiście w formacie mapy bitowej).

Usunięto: 34.

Komentarz: Czy jest to rysunek czy tabela?

3. Zaprojektujmy formularz, w którego skład wchodzić będzie komponent typu `TToolBar`, 7 przycisków `TSpeedButton`, okno edycji `TRichEdit`, przycisk typu `TButton`, menu `TMainMenu`, komponenty `TSaveDialog` i `TOpenDialog`, komponent `TImageList` oraz dwa komponenty `TActionList`. Sposób ich rozmieszczenia na formularzu pokazany jest na rysunku 6.8.

Rys. 6.8. Sposób rozmieszczenia komponentów na formularzu projektu Projekt07.bpr



4. Najpierw na formularzu umieśćmy komponent `TToolBar`, zaś bezpośrednio na nim kolejno komponenty `TSpeedButton`. Posługując się inspektorem obiektów ich cechy `Name` zmieńmy odpowiednio na FileNew, FileOpen, FileSave, Cut, Copy, Paste, Undo.
5. Korzystając z właściwości `Glyph`, rozwińmy opcję `TBitmap` i umieśćmy na każdym z tych przycisków odpowiednią mapę bitową, tak jak na rys. 6.8.
6. Każdemu z naszych komponentów przyporządkujmy funkcję obsługi odrębnego zdarzenia według poniższego schematu:

Usunięto: R

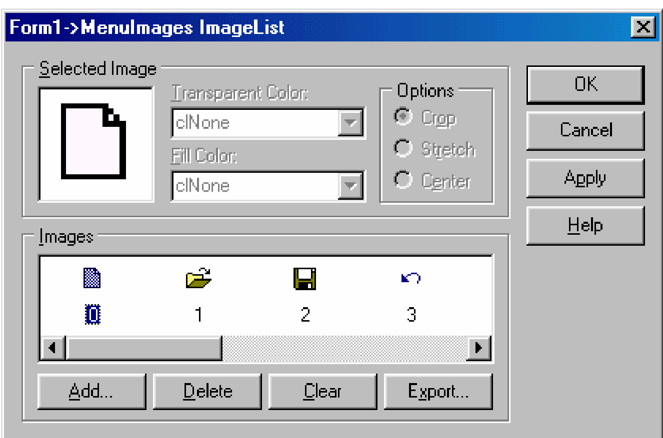
Usunięto: e

```
//--------------------------------------------------------------------
void __fastcall TForm1::FileNewClick(TObject *Sender)
{
     CheckFileSave();
     RichEdit1->Lines->Clear();
     RichEdit1->Modified = FALSE;
}
//--------------------------------------------------------------------
void __fastcall TForm1::FileOpenClick(TObject *Sender)
{
     CheckFileSave();

     if (OpenDialog1->Execute())
       {
        RichEdit1->Lines->LoadFromFile(OpenDialog1->FileName);
        RichEdit1->Modified = FALSE;
        RichEdit1->ReadOnly =
                          OpenDialog1->Options.Contains(ofReadOnly);
       }

}
```

```
//---------------------------------------------------------------------
void __fastcall TForm1::FileSaveAsClick(TObject *Sender)
{
     if (SaveDialog1->Execute())
       {
        RichEdit1->Lines->SaveToFile(SaveDialog1->FileName);
        RichEdit1->Modified = False;
       }
}
//---------------------------------------------------------------------
void __fastcall TForm1::UndoClick(TObject *Sender)
{
    if (RichEdit1->HandleAllocated())
       SendMessage(RichEdit1->Handle, EM_UNDO, 0, 0);
}
//---------------------------------------------------------------------
void __fastcall TForm1::CutClick(TObject *Sender)
{
    RichEdit1->CutToClipboard();
}
//---------------------------------------------------------------------
void __fastcall TForm1::PasteClick(TObject *Sender)
{
    RichEdit1->PasteFromClipboard();
}
//---------------------------------------------------------------------
void __fastcall TForm1::CopyClick(TObject *Sender)
{
    RichEdit1->CopyToClipboard();
}
//---------------------------------------------------------------------
```

7. Cechę Name komponentu ImageList1 zmieńmy na MenuImages. Klikając go dwukrotnie, wczytajmy kolejno potrzebne nam obrazki w postaci map bitowych, każdemu z nich automatycznie powinien zostać przyporządkowany kolejny numer:

Usunięto: na nim



Rys. 6.9. Sposób posługiwania się komponentem TToolBarImages

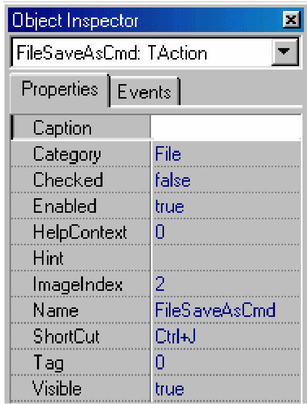
8. Cechę Images (inspektor obiektów, karta Properties) komponentów ActionList1 oraz ActionList2 ustawmy jako MenuImages.
9. Klikając dwukrotnie ActionList1 dostajemy się do pola edycji komponentu. Wybierając New Action zmieniamy kategorię (Categories) na File. Zaznaczając File dostajemy się do okna akcji Actions, zmieniamy Action1 na FileNewcmd, przypisujemy temu komponentowi zerowy indeks obrazka (ImageIndex 0), zaś w opcji Events zdarzeniu OnExecute przypisujemy FileNewClick(). Podobne działania trzeba przeprowadzić z innymi obiektami akcji, tak jak pokazuje to rys. 6.10.

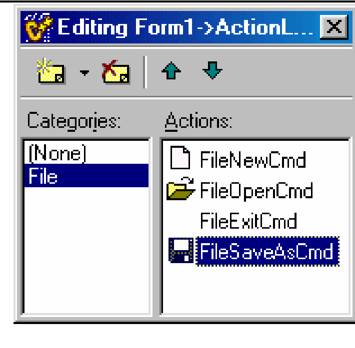
Usunięto: na

Usunięto: jego

Usunięto: któremu

Usunięto: ie postąpmy

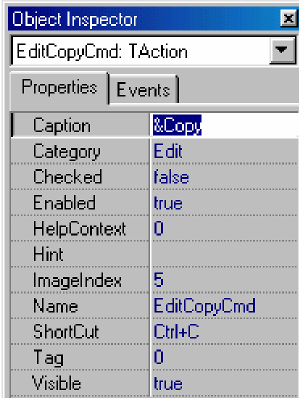


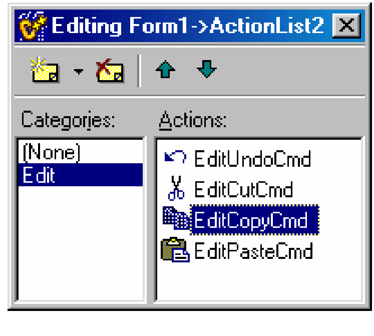


Rys. 6.10. Ustalenie sposobu przypisań właściwości dla komponentów kategorii File

10. W ten sam sposób akcji `FileExitcmd` przypisujemy funkcje obsługi zdarzenia `CloseApplicationClick()`,skojarzonej z przyciskiem `Button1`, którego cechę `Name` zmieniliśmy na `CloseApplication`, zaś cechę `Caption` na &Zamknij.

11. Analogicznie projektujemy właściwości komponentów kategorii `Edit`, ukrywającej się w `ActionList2`, tak jak pokazuje to rysunek 6.11.

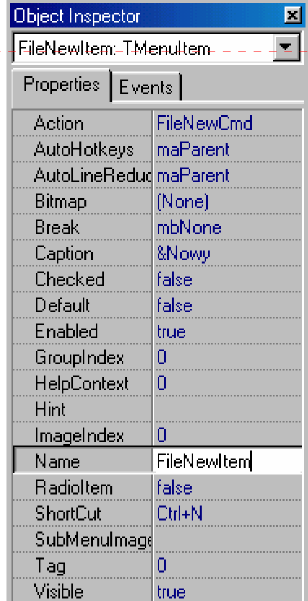


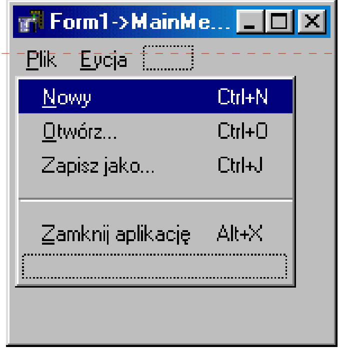


Rys. 6.11. Ustalenie sposobu przypisań właściwości dla komponentów kategorii Edit

12. Przechodzimy do zaprojektowania głównego menu. W karcie właściwości inspektora obiektów, cesze `Images` komponentu `TMainMenu` przypiszmy `MenuImages`.
13. Główne menu składać się będzie z dwóch opcji Plik oraz Edycja. Menu Plik zaprojektujemy w sposób pokazany na rysunku 6.12.

Rys. 6.12. Menu Plik wraz z odpowiednimi przypisaniami w inspektorze obiektów





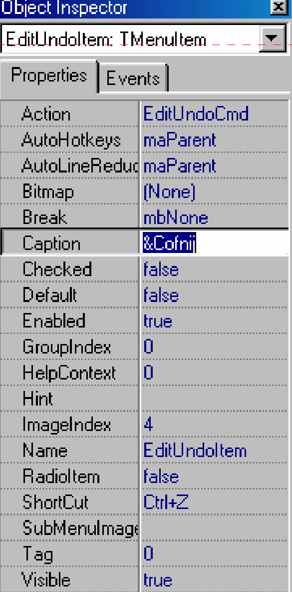
Usunięto: w raz

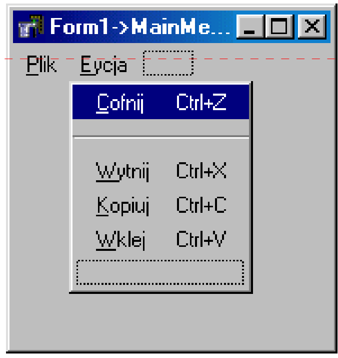
Jeżeli zechcemy, aby w odpowiedzi na wybranie opcji `Nowy` wywoływana była funkcja obsługi zdarzenia `FileNewClick()`, zdarzeniu `OnClick` w karcie zdarzeń należy właśnie przypisać `FileNewClick`.

Usunięto: w karcie zdarzeń,

14. Z kolei menu Edycja zaprojektujemy według przepisu pokazanego na rysunku 6.13.

Rys. 6.13. Menu Edycja wraz z odpowiednimi przypisaniami w inspektorze obiektów





Usunięto: w raz

Na wydruku 6.4 zamieszczono kompletny kod aplikacji *Projekt07.bpr*. W funkcji `FormCreate()` wykorzystaliśmy właściwości `InitialDir` oraz `Filter` obiektów `TOpenDialog` i `TSaveDialog`, z których pierwsza pozwala już w momencie uruchomienia aplikacji ustalić właściwą ścieżkę dostępu do aktualnego katalogu, z kolei druga z wymienionych zapewnia

możliwość odczytania plików posiadających wymagane przez nas rozszerzenia. W tej samej funkcji umieściliśmy również „dymki podpowiedzi” do poszczególnych przycisków, korzystając z właściwości `Hint` oraz `ShowHint`. Śledząc poniższy wydruk zauważymy też, że aby komponenty `TOpenDialog` i `TsaveDialog`, niewidoczne przecież w trakcie uruchomienia programu, generowały zdarzenia polegające na wyświetleniu odpowiednich okien dialogowych, należy w funkcjach odpowiednich zdarzeń skorzystać z metody `Execute()`. Plik z dysku odczytujemy korzystając z metody `LoadFromFile()`, zapisujemy zaś przy pomocy `SaveToFile()`.

W funkcji `CheckFileSave()` skorzystaliśmy z właściwości `Modified` komponentów edycyjnych, w tym również klasy `TRichEdit`. Jeżeli wykonamy jakąkolwiek modyfikację okna edycji, nastąpi wywołanie metody:

```
virtual void __fastcall Modified(void) = 0 ;
```

którą należy obsłużyć, chociażby w sposób zaprezentowany poniżej. Jeżeli zdecydujemy się zapisać zmiany, zostanie wywołana funkcja obsługi zdarzenia `FileSaveAsClick(this)`, w przeciwnym wypadku nastąpi wywołanie funkcji `Abort()` wstrzymującej wykonywania bieżącego zdarzenia.

W języku C++ istnieje słowo kluczowe `this`, będące ważnym elementem wielu tzw. „przeładowywanych operatorów”. Każda funkcja składowa aplikacji lub ogólnie obiektu w momencie wywołania uzyskuje automatycznie wskaźnik do obiektu, który ją wywołał. Dostęp do tego wskaźnika uzyskuje się dzięki słowu (wskaźnikowi) `this`, który jest niejawnym parametrem wszystkich funkcji wchodzących w skład obiektu (aplikacji).
Jeżeli w pewnej, wydzielonej części aplikacji, np. w jakiejś funkcji, wywołujemy funkcję obsługi zdarzenia, której argumentem jest z reguły wskaźnik `TObject *Sender`, należy wówczas jawnie uzyskać do niego dostęp. Z reguły robimy to korzystając właśnie ze wskaźnika `this`.

Wydruk 6.4. Kod modułu *Unit07.cpp* aplikacji wykorzystującej listę akcji `TActionList` w celu zorganizowania pracy głównego menu oraz całego programu.

```
#include <vcl.h>
#pragma hdrstop
#include "Unit07.h"
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;

//----------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//----------------------------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{
     OpenDialog1->InitialDir = ExtractFilePath(ParamStr(0));
     OpenDialog1->Filter =
     "*.dat , *.txt, *.cpp, *.c, *.rtf | *.dat; *.txt; *.cpp;
      *.c; *.rtf";
     SaveDialog1->InitialDir = OpenDialog1->InitialDir;
     SaveDialog1->Filter = "*.*|*.*";

     RichEdit1->ScrollBars = ssVertical;

     FileNew->Hint = "Nowy plik Ctrl+N";
     FileNew->ShowHint = TRUE;
     FileOpen->Hint = "Otwórz plik Ctrl+O";
     FileOpen->ShowHint = TRUE;
     FileSave->Hint = "Zapisz jako... Ctrl+J";
```

```
     FileSave->ShowHint = TRUE;
     Copy->Hint = "Kopiuj Ctrl+C";
     Copy->ShowHint = TRUE;
     Paste->Hint = "Wklej Ctrl+V";
     Paste->ShowHint = TRUE;
     Cut->Hint =   "Wytnij Ctrl+X";
     Cut->ShowHint = TRUE;
     Undo->Hint =   "Cofnij Ctrl+Z";
     Undo->ShowHint = TRUE;

}
//----------------------------------------------------------------------
void __fastcall TForm1::CheckFileSave(void)
{
     if (RichEdit1->Modified)
       {
        switch(MessageBox(NULL, "Zawartość pliku lub okna została"
                          " zmieniona. Zapisać zmiany?", "Uwaga!",
                          MB_YESNOCANCEL | MB_ICONQUESTION))
         {
           case ID_YES    : FileSaveAsClick(this);
           case ID_CANCEL : Abort();
         };
     }
}
//----------------------------------------------------------------------
void __fastcall TForm1::FileNewClick(TObject *Sender)
{
     CheckFileSave();
     RichEdit1->Lines->Clear();
     RichEdit1->Modified = FALSE;
}
//----------------------------------------------------------------------
void __fastcall TForm1::FileOpenClick(TObject *Sender)
{
     CheckFileSave();

     if (OpenDialog1->Execute())
       {
        RichEdit1->Lines->LoadFromFile(OpenDialog1->FileName);
        RichEdit1->Modified = FALSE;
        RichEdit1->ReadOnly =
                         OpenDialog1->Options.Contains(ofReadOnly);
       }

}
//----------------------------------------------------------------------
void __fastcall TForm1::FileSaveAsClick(TObject *Sender)
{
     if (SaveDialog1->Execute())
       {
        RichEdit1->Lines->SaveToFile(SaveDialog1->FileName);
        RichEdit1->Modified = False;
       }
}
//----------------------------------------------------------------------
void __fastcall TForm1::UndoClick(TObject *Sender)
{
    if (RichEdit1->HandleAllocated())
       SendMessage(RichEdit1->Handle, EM_UNDO, 0, 0);
}
//----------------------------------------------------------------------
void __fastcall TForm1::CutClick(TObject *Sender)
{
    RichEdit1->CutToClipboard();
}
//----------------------------------------------------------------------
void __fastcall TForm1::PasteClick(TObject *Sender)
{
    RichEdit1->PasteFromClipboard();
}
//----------------------------------------------------------------------
void __fastcall TForm1::CopyClick(TObject *Sender)
{
```

```
        RichEdit1->CopyToClipboard();
}
//---------------------------------------------------------------------
void __fastcall TForm1::CloseApplicationClick(TObject *Sender)
{
    switch(MessageBox(NULL, " Działanie aplikacji zostanie"
          " zakończone.", "Uwaga!", MB_YESNOCANCEL | MB_ICONQUESTION))
        {
          case ID_YES :
            {
             if (RichEdit1->Modified)
               CheckFileSave();
             Application->Terminate();
            }
          case ID_CANCEL : Abort();
        };
}
//---------------------------------------------------------------------
```

Funkcje zdarzeniowe `CutClick()`, `PasteClick()`, `CopyClick()`, towarzyszące podmenu `Edycja` oraz zaimplementowane w odpowiednich przyciskach zgrupowanych w panelu `ToolBar1`, czyli Cut, Paste, Copy korzystają z metod `RichEdit1->CutToClipboard()`, `RichEdit1->PasteFromClipboard()` i `RichEdit1->CopyToClipboard()`. Funkcje te umożliwiają także usuwanie fragmentu tekstu, wstawianie fragmentu tekstu znajdującego się w schowku (ang. *clipboard*) oraz kopiowanie fragmentu tekstu do schowka. Możliwe jest również zaznaczanie całości tekstu przy wykorzystaniu metody `RichEdit1->SelectAll()`. Aby powtórzyć ostatnio wykonaną (na tekście) operację, należy skorzystać z metody `RichEdit1->HandleAllocated()`, umieszczonej w funkcji obsługi zdarzenia `UndoClick()`.

Usunięto: :
Usunięto: ,
Usunięto: Z
Usunięto: apewniając
Usunięto: możliwość
Usunięto: usunięcia
Usunięto: e
Usunięto: s
Usunięto: e

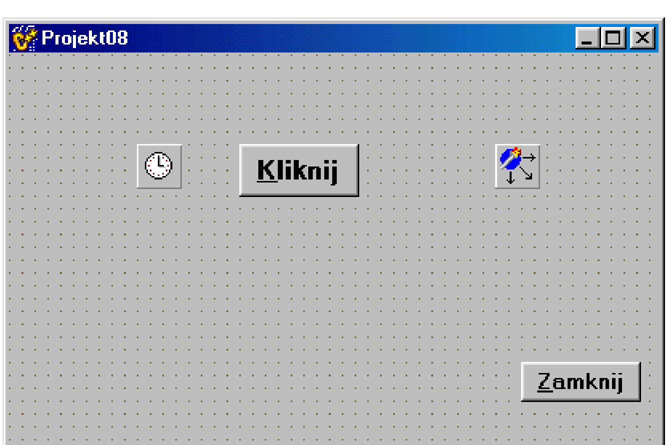
Przykład wykorzystania komponentów TApplicationEvents oraz TTimer

Ćwiczenie 6.5.

1. Załóżmy na dysku oddzielny katalog i nazwijmy go *\Projekt08.*
2. Zaprojektujmy formularz w ten sposób, aby składał się z dwóch komponentów `TButton` oraz po jednym `TTimer` i `TApplicationEvents`, tak jak pokazuje to rys. 6.14.

Usunięto: ,
Usunięto: e

Rys. 6.14. Komponenty formularza projektu Projekt08.bpr



3. Zaznaczając komponent `Timer1`, w karcie zdarzeń inspektora obiektów jego cechę `OnTimer` ustalmy jako `TimerOnTimer`. W wyniku tego działania Timer generować będzie zdarzenia w mniej więcej równych odstępach czasu, określonych przez jego właściwość `Interval`. Treść funkcji obsługi zdarzenia `TimerOnTimer()` wypełnijmy kodem przedstawionym na wydruku 6.5. Zastosujemy proste funkcje

Usunięto: Zapewni nam to, że
Usunięto: owaliśmy

trygonometryczne `sin()` oraz `cos()` w celu opisania toru ruchu przycisku `Button1` oraz całego formularza. Ponieważ funkcje trygonometryczne w wyniku nie zawsze zwracają liczbę całkowitą (właściwości `Top` oraz `Left` muszą być typu `int`), dlatego rezultat wykonania tych funkcji zostanie zaokrąglony poprzez funkcję `floor()`. Funkcje te wymagają włączenia biblioteki *math.h*. `M_PI` jest predefiniowaną stałą reprezentującą liczbę π.

4. W funkcji obsługi zdarzenia `ApplicationEventsActivate()` uaktywnimy działanie Timera korzystając z jego właściwości `Enabled`. Aby dostać się do wnętrza tej funkcji zaznaczmy komponent `ApplicationEvents1` i w karcie zdarzeń inspektora obiektów jego zdarzeniu `OnActivate` przypiszmy `ApplicationEventsActivate`. Naciskając Enter (lub klikając dwa razy) dostaniemy się do wnętrza funkcji odpowiedniego zdarzenia.
5. W analogiczny sposób zaprojektujmy funkcję obsługi zdarzenia `ApplicationEventsDeactivate()`.
6. W funkcji obsługi zdarzenia `Button1Click()` zainicjujemy generator liczb losowych `Randomize()`, następnie korzystając z funkcji `int random(int num)` losującej liczby z przedziału `<0; num-1>` dokonamy losowego wyboru komunikatu przypisanego odpowiedniemu indeksowi tablicy `text`.

Usunięto: ł
Usunięto: a
Usunięto: a
Usunięto: ujemy
Usunięto: jemy
Usunięto: e
Usunięto: owaliśmy
Usunięto: ujemy

Wydruk 6.5. Kompletny kod modułu *Unit08.cpp* projektu *Projekt08.bpr*

```
#include <vcl.h>
#include <math.h>
#pragma hdrstop
#include "Unit08.h"
#pragma package(smart_init)
#pragma resource "*.dfm"

double counter = 0; // licznik
AnsiString text[10];

TForm1 *Form1;
//--------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//--------------------------------------------------------------------
void __fastcall TForm1::TimerOnTimer(TObject *Sender)
{
   counter ++;
   Button1->Top = floor(cos(M_PI*counter/128)*100+100);
   Button1->Left = floor(sin(M_PI*counter/128)*100+100);

   Form1->Top = floor(sin(M_PI*counter/128)*200+200);
   Form1->Left = floor(cos(M_PI*counter/128)*200+200);
}
//--------------------------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{
   Timer1->Enabled = FALSE; // Timer wyłączony
   Timer1->Interval = 1;  // przedział generowania zdarzeń 1 ms

   text[0] = "Bardzo dobrze";
   text[1] = "Za mocno !!!";
   text[2] = "Jeszcze raz";
   text[3] = "Prawie Ci się udało";
   text[4] = "Nie tak !!! Słabiej !!!";
}
//--------------------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{
   Timer1->Enabled = FALSE;
   Application->Terminate();
}
//--------------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
```

```
{
   Randomize();
   ShowMessage(text[random(5)]);
}
//----------------------------------------------------------------------
void __fastcall TForm1::ApplicationEventsActivate(TObject *Sender)
{
     ShowMessage(" Zdarzenie aktywowane");
     Timer1->Enabled = TRUE;
}
//----------------------------------------------------------------------
void __fastcall TForm1::ApplicationEventsDeactivate(TObject *Sender)
{
   Timer1->Enabled = FALSE;
}
//----------------------------------------------------------------------
```

Działanie programu sprawdza się do prostej animacji jednego z przycisków oraz całego formularza. Aby wstrzymać działanie aplikacji należy kliknąć jakieś miejsce na pulpicie poza obszarem formularza.

Usunięto: gdzieś

Ćwiczenie do samodzielnego wykonania

Ćwiczenie 6.6.

Zmodyfikuj przedstawiony na wydruku 6.5 program, tak aby można było wykorzystać inne zdarzenia udostępniane przez `TApplicationEvents`. Pożyteczną ściągawkę można znaleźć w katalogu instalacyjnym Buildera *\Examples\AppEvents*.

Karta Win 3.1

Karta Win 3.1 udostępnia listę komponentów stosowanych w starszych, 16-bitowych wersjach C++Buildera. Nie jest zalecane używanie komponentów posiadających swoje odpowiedniki np. w obecnych kartach Win32 czy Data Controls, co zostało zaznaczone w poniższym zestawieniu.

Komentarz: Win32 czy Win 3.2?

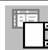
Tabela 6.6. *Komponenty karty Win 3.1*

Ikona	Typ	Znaczenie
	`TTabSet`	Odpowiada komponentowi `TTabControl` z karty Win32.
	`TOutLine`	Odpowiada komponentowi `TTreeView` z karty Win32.
	`TTabbedNoteBook`	Odpowiednik `TPageControl` z karty Win32.
	`TNoteBook`	Odpowiednik `TPageControl`.
	`THeader`	Odpowiada komponentowi `THeaderControl` z karty Win32.
	`TFileListBox`	Komponent dający możliwość wyświetlenia listy plików wskazanego katalogu.
	`TDirectoryListBox`	Element udostępniający listę katalogów wybranego napędu.
	`TDriveComboBox`	Komponent pozwalający dokonać wyboru napędu (stacji dysków).
	`TFilterComboBox`	Komponent, który udostępnia listę plików wyświetlanych z zastosowaniem odpowiedniego filtru. Celowi temu służy właściwość `Mask`.

Usunięto: U

Usunięto: Wybór

Usunięto: U

	DBLookupList	Odpowiada komponentowi TDBLookupListBox z karty Data Controls dostępnej w wersji Enterprise C++Buildera 5.
	DBLLookupCombo	Odpowiada komponentowi TDBLookupComboBox z karty Data Controls dostępnej w wersji Enterprise C++Buildera 5.

Komponenty karty Win 3.1 mimo, iż pochodzą ze starszych wersji C++Buildera, są w dalszym ciągu często używane. Chodzi głównie o komponenty ułatwiające bardzo szybie wczytanie wybranego pliku. Obiekty obsługujące te zdarzenia mają jeszcze jedną poważną zaletę, mianowicie wczytywany plik można natychmiast poddać edycji. Zilustrujemy to na przykładzie prostego ćwiczenia.

Usunięto: no,

Wykorzystanie komponentów TDirectoryListBox, TFileListBox, TFilterComboBox oraz TDriveComboBox

Ćwiczenie 6.7.

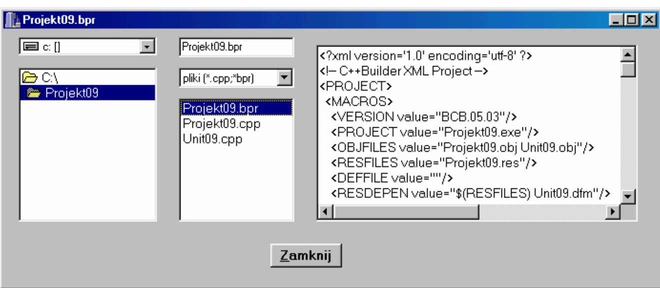
Zaprojektujmy formularz składający się z pojedynczych komponentów TDirectoryListBox, TFileListBox, TDriveComboBox, TFilterComboBox, TEdit, TMemo oraz TButton, tak jak pokazuje to rys. 6.15. Aby obiekty służące do wyboru napędu, wyboru katalogów, przeglądania katalogów „widziały się nawzajem", należy ich cechy odpowiednio ze sobą powiązać. Można to zrobić korzystając z inspektora obiektów lub, co jest bardziej przejrzyste, dokonując odpowiednich przypisań w funkcji FormCreate(), pokazanych na wydruku 6.6.

Usunięto: ,

Usunięto: dokonać

1. Właściwości FileList obiektu DirectoryListBox1 przypiszmy FileListBox1.
2. Właściwości DirList obiektu DriveComboBox1 przypiszmy DirectoryListBox1.
3. Właściwość FileEdit komponentu FileListBox1 ustalmy jako Edit1.
4. Właściwość Filter obiektu FilterComboBox1 ustalmy przykładowo tak, jak pokazuje to poniższy wydruk.
5. Klikając dwukrotnie obszar FileListBox1 dostaniemy się do wnętrza funkcji obsługi zdarzenia FileListBox1Change(). Wystarczy wypełnić ją znanym nam już kodem.
6. Cechę Mask komponentu FileListBox1 ustalmy jako FilterComboBox1->Mask. Czynność tę wykonamy w funkcji obsługi zdarzenia FilterComboBox1Change().

Usunięto: w



Rys. 6.15. Działający Projekt09.bpr

Wydruk 6.6. Kompletny kod modułu *Unit09.cpp* projektu *Projekt09.bpr*

```
#include <vcl.h>
#pragma hdrstop
#include "Unit09.h"
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------
```

```
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//-------------------------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    DirectoryListBox1->FileList = FileListBox1;
    FilterComboBox1->Filter = "wszystkie pliki (*.*)|*.*|"
    "pliki (*.h;*.hpp)|*.h;*.hpp |pliki (*.cpp;*bpr)|*.cpp;*.bpr" ;

    DriveComboBox1->DirList = DirectoryListBox1;
    FileListBox1->FileEdit = Edit1;

    Memo1->ScrollBars = ssBoth;
}
//-------------------------------------------------------------------
void __fastcall TForm1::FilterComboBox1Change(TObject *Sender)
{
   FileListBox1->Mask = FilterComboBox1->Mask;
}
//-------------------------------------------------------------------
void __fastcall TForm1::FileListBox1Change(TObject *Sender)
{
  int iFileHandle;
  int iFileLength;
  char *Buffer;
    try {
      Memo1->Lines->Clear();
      iFileHandle = FileOpen(FileListBox1->FileName.c_str(),
                             fmOpenRead);
      iFileLength = FileSeek(iFileHandle, 0, 2);
      FileSeek(iFileHandle, 0, 0);
      Buffer = new char[iFileLength+1];
      FileRead(iFileHandle, Buffer, iFileLength);
      FileClose(iFileHandle);
      Memo1->Lines->Append(Buffer);
      Form1->Caption = Edit1->Text;
      delete [] Buffer;
    }
    catch(...)
    {
     ShowMessage(" Błąd otwarcia pliku. Nie można przydzielić"
                 " wystarczającej ilości pamięci do wczytania"
                 " pliku.");
    }

}
//-------------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Application->Terminate();
}
//-------------------------------------------------------------------
```

Karta Samples

Karta Samples zawiera 9 przykładowych komponentów. Ich kody źródłowe znajdują się w katalogu instalacyjnym Buildera *\EXAMPLES\CONTROLS\SOURCE*. W momencie włączenia tych komponentów do formularza, ich pliki nagłówkowe zostaną dołączone dyrektywą `#pragma link`, która informuje konsolidator o potrzebie dołączenia danego zbioru do pliku wykonawczego programu.

Tabela 6.7. *Komponenty karty Samples*

Ikona	Typ	Znaczenie
	`TPie`	Element służący do przedstawiania okręgu lub wycinka okręgu. Właściwość `Angles` uruchamia Pie Angles Editor. Kod źródłowy komponentu można znaleźć w plikach *piereg.** oraz *pies.**.
	`TTrayIcon`	Komponent, który umożliwia m.in. wykonanie zamiany ikon, w tym ich prostej animacji. Kod źródłowy komponentu można znaleźć w plikach *Trayicon.**.
	`TPerformanceGraph`	Element służący do przedstawienia grafiki. Kod źródłowy komponentu znajduje się w plikach *PERFGRAP.**.
	`TCSpinButton`	Komponent umożliwiający płynne zmniejszanie i zwiększanie zawartości liczbowej wybranego pola edycji. Jego kod źródłowy znajduje się w plikach *CSPIN.**.
	`TCSpinEdit`	Element stanowiący połączenie `TCSpinButton` oraz `TEdit`. Kod źródłowy można znaleźć w plikach *CSPIN.**.
	`TCColorGrid`	Komponent umożliwiający dokonanie wyboru koloru. Jego kod źródłowy znajduje się w plikach *CGRID.**.
	`TCGauge`	Komponent przedstawiający wskaźnik postępu. Dzięki właściwości `Kind` można go przedstawić w postaci paska, liczby, koła lub wycinka koła. Jego kod źródłowy znajduje się w plikach *CGAUGES.**.
	`TCDirectoryOutLine`	Wyświetla drzewo katalogów znajdujących się na dysku. Kod źródłowy komponentu znajduje się w plikach *cdiroutl.**.
	`TCCalendar`	Komponent wyświetlający aktualną datę w postaci uproszczonego kalendarza. Jego kod źródłowy znajduje się w pliku *CCALENDR.**.

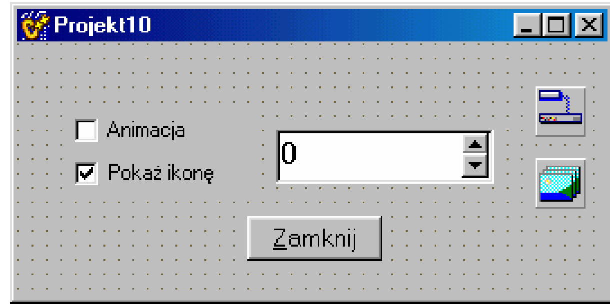
Usunięto: ak
Usunięto: U
Usunięto: U
Usunięto: S
Usunięto: ak

Jako przykład wykorzystania niektórych komponentów z kart Samples oraz Standard stworzymy prostą aplikację, przy pomocy której możliwym będzie animacja ikon. Zmieniające się ikony będą wyświetlane na dolnym pasku zadań w prawym rogu monitora tuż obok zegara.

Wykorzystanie komponentów TCSpinEdit, TTrayIcon, TImageList oraz TCheckBox

Ćwiczenie 6.8.

Zaprojektujmy formularz składający się z pojedynczych komponentów `TCSpinEdit`, `TTrayIcon`, `TImageList`, `TButton` oraz dwóch komponentów typu `TCheckBox` tak jak pokazuje to rysunek. 6.16.



Rys. 6.16. Komponenty formularza projektu Projekt10.bpr

1. Korzystając z inspektora obiektów właściwość `Icons` komponentu `TrayIcon1`

zmieńmy na ImageList1. Tym samym spowodujemy, że ikony wczytane do komponentu ImageList1 będą „widziane" przez TrayIcon1. W podobny sposób (żeby zbytnio nie komplikować dalszych rozważań) właściwościom PopupMenuOn oraz RestoreOn przypiszmy imNone.

2. Cechy Caption komponentów CheckBox1 oraz CheckBox2 zmieńmy odpowiednio na Animacja oraz Pokaż ikonę.

3. Cechy Increment, MinValue oraz MaxValue komponentu CSpinEdit1 ustalmy w sposób pokazany w funkcji FormCreate() na wydruku 6.7.

4. We wnętrzu tej samej funkcji cechę Visible komponentu TrayIcon1 uzależnimy od aktualnego stanu komponentu CheckBox2 reprezentowanego przez właściwość Checked.

5. Właściwość Animate komponentu TrayIcon1 uzależnimy od stanu cechy Checked komponentu CheckBox1.

6. Właściwości AnimateInterval komponentu TrayIcon1 przypiszmy wartość cechy Value komponentu CSpinEdit1.

Kompletny kod głównego modułu naszej aplikacji powinien wyglądać tak jak przedstawiono to na wydruku 6.7.

Wydruk 6.7. Kod modułu *Unit10.cpp* aplikacji *Projekt10.bpr* wykonującej prostą animację ikon.

```
#include <vcl.h>
#pragma hdrstop
#include "Unit10.h"
#pragma package(smart_init)
#pragma link "CSPIN"
#pragma link "Trayicon"
#pragma resource "*.dfm"

TForm1 *Form1;
//--------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//--------------------------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    CSpinEdit1->MaxValue = 2000;
    CSpinEdit1->MinValue = 100;
    CSpinEdit1->Increment = 100;
    TrayIcon1->Visible = CheckBox2->Checked;
}
//--------------------------------------------------------------------
void __fastcall TForm1::CheckBox1Click(TObject *Sender)
{
    TrayIcon1->Animate = CheckBox1->Checked;
    if(CheckBox1->Checked == FALSE)
      TrayIcon1->IconIndex = 0;
    Update();
}
//--------------------------------------------------------------------
void __fastcall TForm1::CheckBox2Click(TObject *Sender)
{
    TrayIcon1->Visible = CheckBox2->Checked;
    CheckBox1->Enabled = CheckBox2->Checked;
    Update();

}
//--------------------------------------------------------------------
void __fastcall TForm1::CSpinEdit1Change(TObject *Sender)
```

```
{
    TrayIcon1->AnimateInterval = CSpinEdit1->Value;
}
//-------------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Application->Terminate();
}
//-------------------------------------------------------------------
```

Przedstawiony algorytm każdy na własny użytek może wzbogacić o szereg innych elementów. Zaglądając do kodów źródłowych poszczególnych komponentów karty Samples możemy samodzielnie odszyfrować jeszcze wiele ich możliwości.

Komponent TCCalendar

Ćwiczenie 6.9.

Zaprojektujemy aplikację wykorzystującą komponent `TCCalendar`. Dodatkowo posłużymy się komponentami `TBitBtn` z karty Additional oraz `TGroupBox` z karty Standard. Wykorzystamy też dobrze nam już znany przycisk `TButton`.

1. Na formularzu umieszczamy komponent `Calendar1` reprezentujący klasę `TCCalendar`. W inspektorze obiektów jego cechę `BorderStyle` zmieńmy na `bsSingle`. Klikając nań dwukrotnie dostajemy się do funkcji obsługi zdarzenia `Calendar1Change()`. Korzystając z metody `DateToStr()` właściwości `Caption` naszego kalendarza przypiszmy aktualną datę `Calendar1->CalendarDate`.

   Usunięto: e
2. Rozmieśćmy na formularzu dwa komponenty `GroupBox1` oraz `GroupBox2` reprezentujące klasę `TGroupBox`. Ich cechy `Caption` zmieńmy odpowiednio na &Cofnij oraz &Dalej.
3. W obszarze wymienionych komponentów rozmieszczamy po dwa obiekty `TBitBtn`. Możemy uprzednio przy pomocy edytora rysunków przygotować odpowiednie rysunki i korzystając z właściwości `Glyph` umieścić je na przyciskach, tak jak pokazuje to rys. 6.17.
4. Korzystamy z metod `PrevYear()`, `PrevMonth()`, `NextYear()` oraz `NextMonth()` w celu uzupełnienia naszego kalendarza w funkcje obsługi odpowiednich zdarzeń polegających na wybraniu kolejnego roku lub miesiąca. Kompletny kod naszej aplikacji znajduje się na wydruku 6.8.

Rys. 6.17. Działający kalendarz

Wydruk 6.8. Kod modułu *Unit11.cpp* kalendarza: *Projekt11.bpr*

```
#include <vcl.h>
#pragma hdrstop
#include "Unit11.h"
#pragma package(smart_init)
#pragma link "CCALENDR"
#pragma resource "*.dfm"

TForm1 *Form1;

//--------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//--------------------------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    Caption = "Aktualna data: Rok-Miesiąc-Dzień   "
              + DateToStr(Calendar1->CalendarDate);
}
//--------------------------------------------------------------------
void __fastcall TForm1::Calendar1Change(TObject *Sender)
{
    Caption = "Rok-Miesiąc-Dzień   "
              + DateToStr(Calendar1->CalendarDate);
}
//--------------------------------------------------------------------
void __fastcall TForm1::BitBtn1Click(TObject *Sender)
{
    Calendar1->PrevYear();
}
//--------------------------------------------------------------------
void __fastcall TForm1::BitBtn2Click(TObject *Sender)
{
    Calendar1->PrevMonth();
}
//--------------------------------------------------------------------
void __fastcall TForm1::BitBtn4Click(TObject *Sender)
{
    Calendar1->NextYear();
}
//--------------------------------------------------------------------
void __fastcall TForm1::BitBtn3Click(TObject *Sender)
{
    Calendar1->NextMonth();
}
//--------------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Application->Terminate();
}
//--------------------------------------------------------------------
```

Wykonany własnymi siłami kalendarz może wzbogacić nasze aplikacje o opcje pozwalające na proste kontrolowanie aktualnej daty bez potrzeby wnikania w skomplikowane zasoby systemu Windows. Oczywiście taki kalendarz w każdej chwili możemy ukryć. Wystarczy jego cechę `Visible` ustalić jako `FALSE`, co wcale nie przeszkadza, aby aktualna data nie była wyświetlana w jakimś miejscu aplikacji.

Karta ActiveX

Komponenty karty ActiveX nie wchodzą w skład biblioteki VCL. Są to przykładowe obiekty ActiveX, zaprojektowane w ten sposób, by można było natychmiast skorzystać z ich usług.

Tabela 6.8. *Niektóre komponenty karty ActiveX*

Ikona	Typ	Znaczenie
	TChartfx	Obiekt ActiveX służący do tworzenia wykresów.
	TVSSpell	Visual Speller Control Properties. Komponent pełniący rolę tzw. *spell-chackera*.
	TF1Book	Obiekt posiadający cechy arkusza kalkulacyjnego.
	TVtChart	Komponent służący to tworzenia wykresów.

Komponent TF1Book

Ćwiczenie 6.10.

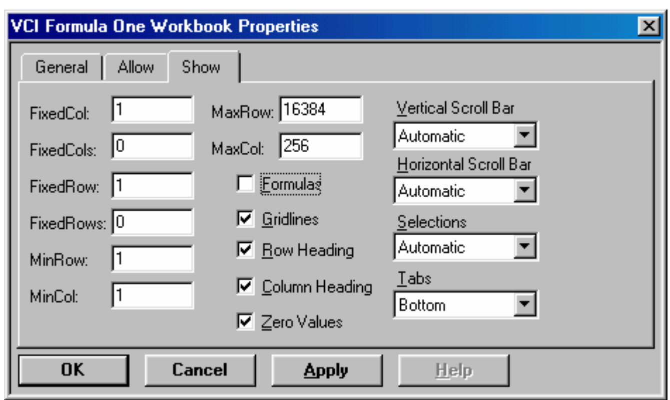
Usunięto: 9.

Jako przykład wykorzystania w naszych aplikacjach niektórych komponentów karty ActiveX pokażemy, jak w bardzo prosty sposób można skorzystać z reprezentanta klasy `TF1Book`.

1. Umieśćmy na formularzu komponent `F1Book1`. Klikając dwukrotnie jego obszar możemy zapoznać się z jego właściwościami, które w większości są zdublowane w analogicznej karcie inspektora obiektów.

Usunięto: w

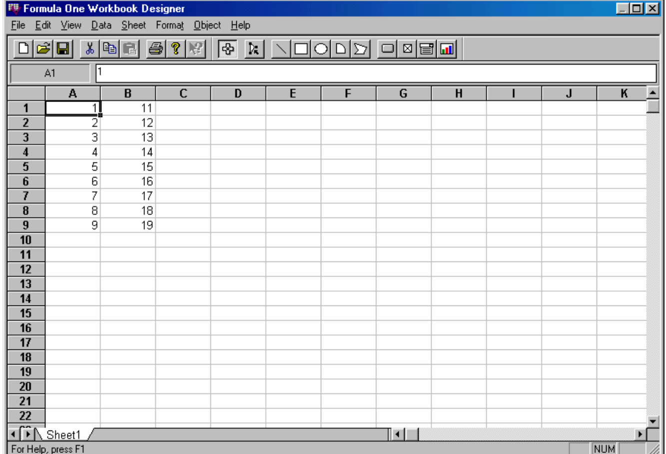
Usunięto: ze



Rys. 6.18. Właściwości VCI Formula One Workbook

2. Po skompilowaniu i uruchomieniu aplikacji dwukrotnie klikając prawym klawiszem myszki dostaniemy się do Formula One Workbook Designer, który jest odpowiednikiem prostego arkusza kalkulacyjnego. Jego obszar podzielony jest na komórki, których współrzędne określone są w sposób typowy dla arkuszy kalkulacyjnych: wiersze (ang. *rows*) i kolumny (ang. *columns*). Korzystając z menu File|Read możemy wczytać dane zawarte w plikach tekstowych, typowe dla formatu Excela, czyli **.xls* lub w formacie **.vts*, który jest preferowanym formatem Workbooka. Dane w postaci liczb możemy też wpisywać samodzielnie. Również zapis danych do pliku nie powinien przedstawiać żadnych problemów.

Rys. 6.19. Formula One Workbook Designer

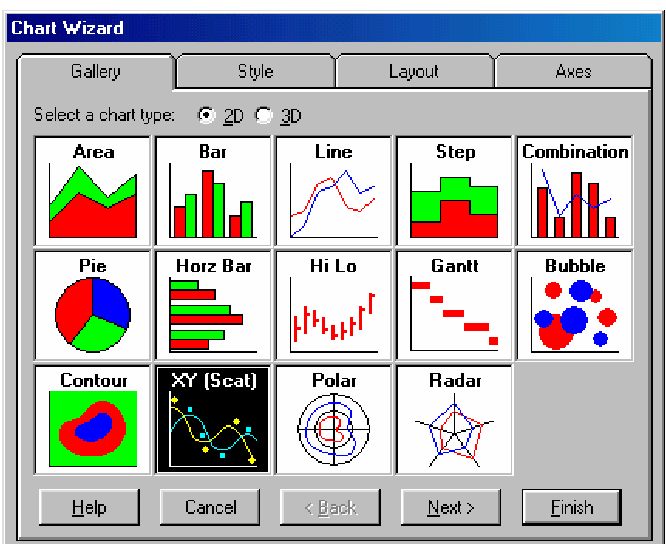


3. Jeżeli zechcemy graficznie zobrazować nasze dane należy najpierw je zaznaczyć albo przy pomocy myszki, albo korzystając z kombinacji klawiszy Shift + strzałka (kursor).

Usunięto: ,

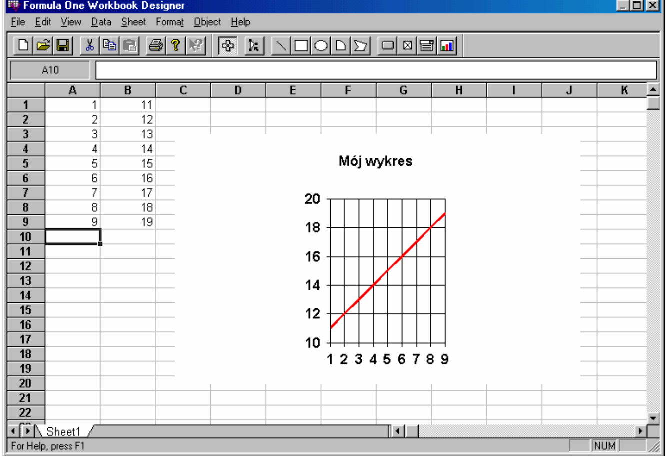
4. Wybieramy ostatni przycisk z podpowiedzią `Chart` (diagram, wykres) i „ciągniemy" myszką na wolnym obszarze arkusza. W ten sposób możemy uaktywnić kreatora wykresów Chart Wizard. Zawiera on bogaty zbiór różnego rodzaju diagramów i wykresów. Zakładki Gallery, Style, Layout oraz Axes ułatwiają optymalny dobór parametrów wykresu oraz dokonania jego opisu.
   Jeżeli dokonamy już wyboru najbardziej odpowiadających nam opcji, kończymy przyciskiem Finish.

Rys. 6.20. Kreator wykresów



5. Końcowy wynik możemy zobaczyć w postaci podobnej do tej przedstawionej na rys. 6.21.

Rys. 6.21. Wykres umieszczony na arkuszu kalkulacyjnym



Karta Internet

Wykorzystując komponenty karty Internet można w aplikacjach umieszczać opcje pozwalające na korzystanie z sieci Internet oraz protokołu TCP/IP.

Tabela 6.9. *Podstawowe komponenty karty Internet*

Ikona	Typ	Znaczenie
	TClientSocket	Komponent ułatwiający połączenie z innym komputerem w sieci.
	TServerSocket	Komponent odpowiadający na żądania innych komputerów w sieci.
	TCppWebBrowser	Komponent wyświetlający stronę HTML w postaci Web. Warunkiem jest posiadanie przeglądarki Internet Explorer wersji 4 lub wyższej.
	TWebDispatcher	Komponent, przy pomocy którego następuje konwersja zwykłego modułu danych na postać Web.
	TPageProducer	Komponent konwertujący szablon HTML na kod, który może być następnie przeglądany.
	TQueryTableProducer	Komponent tworzący tablice HTML na podstawie rekordów obiektu typu TQuery.
	TDataSetTableProducer	Komponent tworzący tablice HTML na podstawie rekordów obiektu typu TDataSet.

Usunięto: W

Usunięto: K

Usunięto: K

Usunięto: e

Usunięto: z

Usunięto: T

Karta Servers

Karta Servers zawiera 30 komponentów będących swego rodzaju wizualizacją aktualnie dostępnych serwerów COM dokonaną na potrzeby biblioteki VCL. Wszystkie wywodzą się z obiektu `TOleServer`. Przy ich pomocy możemy automatycznie połączyć się z wybranym serwerem COM.

Rys. 6.22. Karta Servers

Dokładne omówienie wszystkich komponentów karty Servers wraz z ich właściwościami i metodami, z których korzystają, a tym samym budowy serwerów COM, znacznie wykracza poza ramy naszych ćwiczeń. Niemniej jednak możemy chociażby jakościowo zapoznać się z podstawowymi własnościami wybranych obiektów. Prawdopodobnie nie ma wśród nas nikogo, kto by nie miał do czynienia z narzędziami pakietu Office. Do najbardziej podstawowych aplikacji tego pakietu należy oczywiście zaliczyć Worda, Excela oraz Power Pointa. Spróbujmy zatem, korzystając z bardzo prostych funkcji połączyć się z wymienionymi aplikacjami.

Tabela 6.10. *Podstawowe komponenty karty Servers służące do połączenia z narzędziami pakietu Office*

Ikona	Typ	Znaczenie
	`TPowerPointApplication`	Umożliwia połączenie z aplikacjami Power Point. Komponent niewidzialny. Jego kod źródłowy znajduje się w plikach *PowerPoint_97_SRVR.**. znajdujących się w katalogach *\Include\VCL* oraz *\Source\Comservers\Office97*[5]
	`TWordApplication`	Umożliwia połączenie z aplikacjami Worda. Komponent niewidzialny. Jego kod źródłowy znajduje się w plikach *Word_97_SRVR.**. znajdujących się w katalogach *\Include\VCL* oraz *\Source\Comservers\Office97*
	`TExcelApplication`	Umożliwia połączenie z aplikacjami Excela. Komponent niewidzialny. Jego kod źródłowy znajduje się w plikach *Excel_97_SRVR.**. znajdujących się w katalogach *\Include\VCL* oraz *\Source\Comservers\Office97*

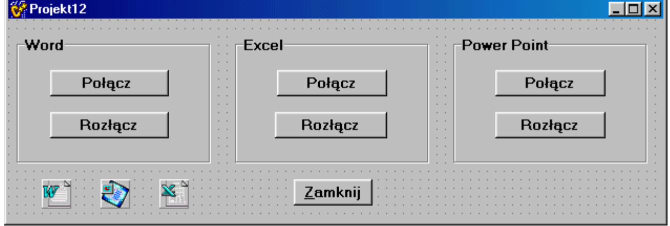
Komponenty TPowerPointApplication, TWordApplication oraz TExcelApplication

Ćwiczenie 6.11.

Zaprojektujemy formularz składający się z trzech komponentów typu `TGroupBox`. W ich obszarach umieszczamy po dwa przyciski `TButton`, które będą odpowiadać za nawiązanie połączenia z wybranym serwerem COM oraz jego dezaktywację. Dodatkowo na formularzu umieszczamy po jednym komponencie `TPowerPointApplication`, `TWordApplication` oraz `TExcelApplication`, podobnie jak na rysunku 6.23.

[5] Nazwa ostatniego katalogu będzie pochodzić od nazwy katalogu, w którym zainstalowana jest wybrana wersja pakietu Office. Wersję pakietu Office, z którą chcemy współpracować należy podać w trakcie instalacji Borland C++Buildera 5.

Usunięto: 0.



Rys. 6.23. Aplikacja wykorzystująca przykładowe komponenty karty Servers w celu dokonania połączeń z wybranymi serwerami COM

1. W najprostszym przypadku ustanowienie połączenia z wybranym serwerem dokonujemy korzystając z metody `Connect()`, której odpowiednie wywołania zostały umieszczone w funkcji `FormCreate()`.
2. W celu wizualizacji połączenia przeważnie korzystamy z właściwości `Visible` poszczególnych obiektów. Pewnym wyjątkiem mogą być aplikacje Excela, gdzie niekiedy wygodniej jest skorzystać z metody `Set_Visible()` wywoływanej z odpowiednimi parametrami.
3. Aby bezpiecznie dezaktywować połączenie korzystamy z metody `Disconnect()`.

Wydruk 6.9. Kod modułu *Unit12.cpp* aplikacji realizującej połączenia z wybranymi serwerami COM.

```
#include <vcl.h>
#pragma hdrstop
#include "Unit12.h"
#pragma package(smart_init)
#pragma link "PowerPoint_97_SRVR"
#pragma link "Word_97_SRVR"
#pragma link "Excel_97_SRVR"
#pragma resource "*.dfm"

TForm1 *Form1;

//--------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//--------------------------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    // PowerPointApplication1->InitServerData();
    PowerPointApplication1->Connect();

    // ExcelApplication1->InitServerData();;
    ExcelApplication1->Connect();

    // WordApplication1->InitServerData();
    WordApplication1->Connect();
}
//--------------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    WordApplication1->Visible = TRUE;
}
//--------------------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{
    WordApplication1->Disconnect();
}
//--------------------------------------------------------------------
void __fastcall TForm1::Button4Click(TObject *Sender)
{
    PowerPointApplication1->Visible = TRUE;
}
//--------------------------------------------------------------------
void __fastcall TForm1::Button5Click(TObject *Sender)
{
```

```
    PowerPointApplication1->Disconnect();
}
//----------------------------------------------------------------------
void __fastcall TForm1::Button6Click(TObject *Sender)
{
    ExcelApplication1->Set_Visible(0, 1);
}
//----------------------------------------------------------------------
void __fastcall TForm1::Button7Click(TObject *Sender)
{
    ExcelApplication1->Disconnect();
}
//----------------------------------------------------------------------
void __fastcall TForm1::Button3Click(TObject *Sender)
{
    Application->Terminate();
}
//----------------------------------------------------------------------
```

Metoda `Disconnect()` wchodzi w zakres innej metody `BeforeDestruction()`, której wywoływanie w sposób jawny nie jest jednak zalecane.

Korzystając z prostego algorytmu przedstawionego na wydruku 6. 9 możemy bez problemu z poziomu działającej aplikacji napisanej w Borland C++Builderze 5 uruchomić wymienione narzędzia pakietu Office. Będą one pracowały w taki sam sposób jakby były uruchamiane w sposób tradycyjny, tzn. bezpośrednio z pulpitu. Pokazana idea komunikacji COM pozwala też na wielokrotne uruchamianie i dezaktywację wybranego połączenia.

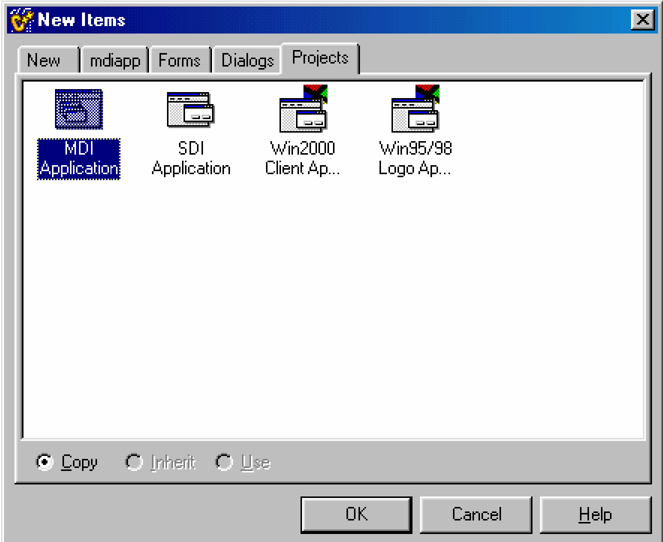
Podsumowanie

W trakcie niniejszego rozdziału zapoznaliśmy się z podstawowymi, dostępnymi w C++Builderze 5 elementami biblioteki VCL. Wykonując parę prostych ćwiczeń nauczyliśmy się posługiwać właściwościami, zdarzeniami oraz metodami różnych komponentów. Przykładowe, kompletne wydruki stosowanych algorytmów pomogą nam zrozumieć, jak z pokazanych komponentów możemy skorzystać w praktyce.

Rozdział 7 .
Aplikacje SDI oraz MDI

Wszystkie przedstawione w poprzednich rozdziałach przykładowe aplikacje w każdym szczególe konstruowaliśmy samodzielnie nabierając wprawy w manipulowaniu komponentami biblioteki VCL. Należy jednak pamiętać, że istnieje dużo prostszy (ale mniej kształcący) sposób zaprojektowania formularza. Można w tym celu skorzystać z menu File|New|Projects. W ten sposób dostaniemy się do zakładki z gotowymi szablonami formularza (rys. 7.1).

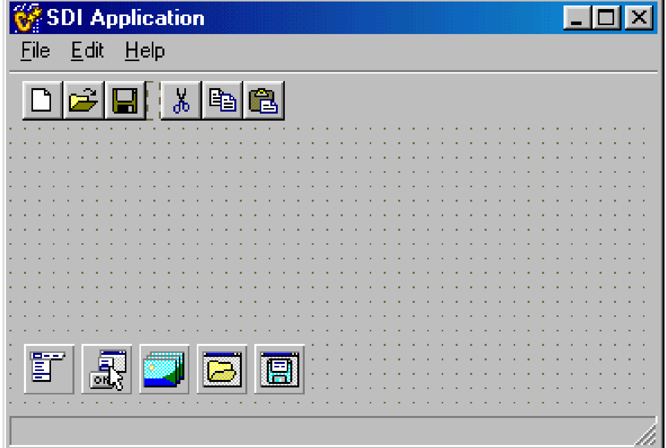
Rys. 7.1. Dostępne, przykładowe projekty różnych formularzy



Aplikacje jednodokumentowe

Wybierając przykładowy projekt aplikacji jednodokumentowej SDI Application (ang. *Single Document Interface*) otrzymujemy gotowy do użycia i ewentualnie dalszego uzupełniania formularz. Widzimy na nim gotowe komponenty `TSaveDialog`, `TOpenDialog`, `TImageList`, `TActionList`, `TMainMenu`, `TStatusBar`. Wszystkie mają już odpowiednio wykonane przypisania. Aplikacja taka, przynajmniej w swej warstwie edycyjnej jest praktycznie gotowa do użycia.

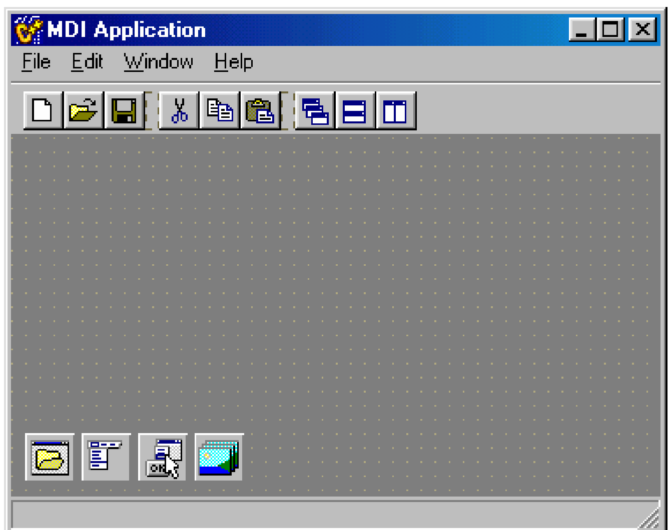
Rys. 7.2. Formularz aplikacji jednodokumentowej



Aplikacje wielodokumentowe

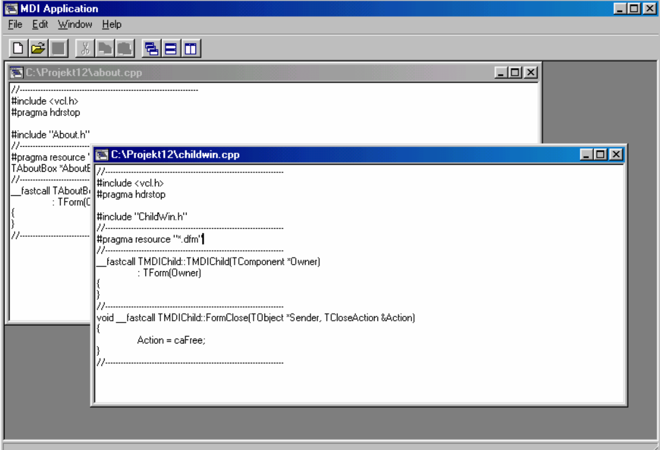
Aplikacje wielodokumentowe MDI Application (ang. *Multi Document Interface*) służą do zarządzania zdarzeniami zachodzącymi w kilku oknach jednocześnie. Podstawową rzeczą, jaka odróżnia je od aplikacji SDI, jest styl stosowanego formularza. O ile w przypadku aplikacji SDI styl formularza reprezentowany przez właściwość `FormStyle` jest typu `fsNormal` (zob. karta właściwości inspektora obiektów), to w przypadku aplikacji MDI formularz posiadać będzie styl `fsMDIForm`. Wszystkie jego okna potomne reprezentowane będą przez formularze `fsMDIChild`. Centralną część formularza widocznego na rysunku 7.3 stanowi tzw. okno klienta (ang. *client window*).

Rys. 7.3. Formularz aplikacji MDI



Wszystkie okna potomne (ang. *child window*) umieszczane w oknie klienta, są całkowicie mu podporządkowane, tak jak pokazuje to rysunek 7.4

Usunięto: będąc



Rys. 7.4. Przykład aplikacji korzystającej z wielu dokumentów wyświetlanych w oknach potomnych

Okna takie możemy dowolnie konfigurować korzystając z przycisków znajdujących się na pasku menu lub bezpośrednio z głównego menu (menu systemowego).

Zachęcam Czytelników do samodzielnego przetestowania właściwości przedstawionych formularzy, a także do zaznajomienia się z ich kodami źródłowymi. Będzie to ciekawe i kształcące zajęcie. Również każda próba uzupełnienia formularzy o nowe, własne elementy pozwoli nam w większym stopniu oswoić się z tego typu aplikacjami.

Usunięto: dodatkowo ciekawym i kształcącym zajęciem niewątpliwie byłoby w ramach samodzielnego ćwiczenia zaznajomienie się z kodami źródłowymi tych formularzy.

Usunięto: ich

Podsumowanie

Celem tego krótkiego, kończącego już książkę, rozdziału było zaprezentowanie gotowych do użycia projektów formularzy udostępnianych nam przez C++Buildera. Dodatkowo każdy z Czytelników mógł się zapoznać z szablonem aplikacji klienta do Windows 2000 lub z techniką tworzenia logo w aplikacjach. Zamieszczone w menu File|New|Projects przykłady prawie już gotowych programów należy jednak traktować jako swego rodzaju pomoc metodyczną. Największym bowiem błędem, jaki może zrobić osoba zaczynająca dopiero tworzyć w C++Builderze jest bezkrytyczne sięgnięcie do omówionych przykładów. Pokazane przykładowe aplikacje mogą być nam bardzo pomocne pod jednym tylko warunkiem, jakim jest świadomość, że coś podobnego jesteśmy w stanie stworzyć również samodzielnie.

Usunięto: może

Usunięto: , już na wstępie

Usunięto: mianowicie, jeżeli będziemy mieli