helion.pl
KSIĘGARNIA INTERNETOWA WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

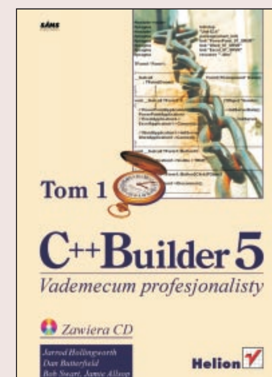
TWÓJ KOSZYK

DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

C++ Builder. Vademecum profesjonalisty. Tom I



Autor: Jarrod Hollingworth, Dan Butterfield, Bob Swart, Jamie Allsop
Tłumaczenie: zbiorowe
ISBN: 83-7197-447-7
Tytuł oryginału: C++ Builder 5 Developer's Guide
Format: B5, stron: 1040
oprawa twarda
Zawiera CD-ROM

To długo oczekiwana książka (poprzednie jej wydanie opisywało wersję 3.)! Jest obszernym przewodnikiem, opisującym możliwości C++Buildera 5. Znajdziesz w niej opis takich mechanizmów, jak: InternetExpress, ADOExpress, Interbase Express, TeamSource, CodeGuard i wielu innych. Książka opisuje także metody wykorzystywania zaawansowanych właściwości COM+, MIDAS-a i specyfikacji CORBA.
Jeżeli używasz C++Buildera 5 w wersji Standard, Professional lub Enterprise, dzięki tej książce możesz osiągnąć mistrzostwo w programowaniu. Od zaawansowanych programistów dowiesz się, jak efektywnie tworzyć aplikacje typu desktop, aplikacje internetowe i rozproszone systemy bazodanowe.

- Naucz się efektywnych technik śledzenia programów przy użyciu CodeGuarda, umożliwiającego wykrywanie przyczyn wielu błędów, między innymi wycieków pamięci i gubienia zasobów.
- Doprowadź do mistrzostwa swoje techniki tworzenia wielowarstwowych aplikacji za pomocą MIDAS-a.
- Odkryj zaawansowane możliwości, tkwiące w najnowszych rozwiązaniach multimedialnych, wykorzystujących: OpenGL, DirectX i formaty AVI, MPEG i MP3.
- Naucz się tworzyć własne komponenty za pomocą specjalistycznych edytorów komponentów i edytorów właściwości.
- Poznaj techniki programowania rozproszonego i tworzenia skalowalnych aplikacji na bazie technologii: COM, COM+, CORBA i DCOM.
- Zobacz, w jaki sposób możesz zwiększyć efektywność swoich aplikacji dzięki optymalizacjom projektów, algorytmów i zarządzania pamięcią.
- Wykorzystaj powszechnie stosowane techniki pakietowania, dystrybucji i ochrony swych aplikacji.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

O Autorach **19**

Wprowadzenie **29**

Przedmowa do wydania polskiego **35**

Rozdział 1. Wprowadzenie do C++Buildera **39**

Podstawowe elementy C++Buildera 39
- Pierwsze spojrzenie 40
- Pierwsze programy 45
- Kilka pytań 52

Co nowego w wersji 5. C++Buildera? 53
- Programowanie internetowe 54
- Aplikacje rozproszone 54
- Projektowanie zespołowe 55
- Lokalizacja aplikacji 55
- Śledzenie 55
- Programowanie bazodanowe 56
- Produktywność programisty 56

Zgodność i unowocześnianie aplikacji 57
- Unowocześnianie wersji C++Buildera 57
- Obsługa istniejących projektów 57
- Tworzenie projektów zgodnych z poprzednimi wersjami C++Buildera 58
- Inne problemy ze zgodnością 58

Migracja ze środowiska Delphi 59
- Komentarze 59
- Zmienne 59
- Stałe 60
- Operatory 61
- Przepływ sterowania w programie 64
- Funkcje i procedury 66
- Klasy 67
- Dyrektywy preprocesora 69
- Typy plików 70

Mocne i słabe strony C++Buildera 71
- Wizualna rzeczywistość — naprawdę błyskawiczne tworzenie aplikacji 71
- Dotrzymywanie kroku — zgodność ze standardami 73

Osobiste preferencje ... 74
Zalety i wady C++Buildera — konkluzje ... 74
Przygotowania do Kyliksa ... 75
Podobieństwa pomiędzy Kyliksem a C++Builderem ... 75
Różnice pomiędzy Kyliksem a C++Builderem ... 76
Podsumowanie ... 77

Rozdział 2. Programowanie w C++Builderze ... 79

Style kodowania a czytelność programu ... 79
Proste i zwięzłe kodowanie ... 79
Akapitowanie kodu ... 81
Sugestywne nazewnictwo elementów programu ... 86
Właściwe konstruowanie kodu ... 92
Używanie komentarzy ... 93
Zalecane praktyki programistyczne ... 97
Obsługa wyjątków ... 109
Zarządzanie pamięcią operacyjną przy użyciu operatorów new i delete ... 114
Rzutowanie typów w stylu C++ ... 117
Używanie preprocesora ... 119
Wykorzystanie standardowej biblioteki C++ ... 121
Podsumowanie ... 122

Rozdział 3. Interfejs użytkownika ... 123

Podstawowe zasady konstrukcji interfejsu użytkownika ... 123
Przykładowe projekty wykorzystywane w tym rozdziale ... 126
Kalkulator — wprowadzenie do projektu ... 127
Zwiększanie użyteczności aplikacji drogą sprzężenia zwrotnego ... 128
Wykorzystanie komponentów TProgressbar i TCGauge ... 128
Wygląd kursora ... 129
Wykorzystanie paska statusu TStatusBar ... 131
Podpowiedzi kontekstowe ... 139
Kontrola migracji skupienia pomiędzy elementami interfejsu ... 156
Dbałość o wygląd interfejsu ... 164
Symbole na przyciskach ... 165
Grupowanie przycisków ... 166
Słowo na temat migotania ... 167
Wzbogacanie tekstu symbolami ... 167
Kolorystyka interfejsu ... 171
Użycie nieprostokątnych okien ... 171
Konfigurowalność interfejsu ... 173
Dokowanie ... 173
Zmiana rozmiarów kontrolek ... 177
Wykorzystanie paska kontrolnego TControlBar ... 186
Kontrolowanie widoczności obiektów interfejsu ... 196
Indywidualizacja ustawień ... 200

Zróżnicowane konfiguracje graficzne ... 208
Różnice w rozdzielczości ekranu ... 208
Różnice w wielkościach czcionek ... 209
Różnice w liczbie kolorów ... 209
Techniki łagodzące złożoność konstrukcji interfejsu ... 210
Scentralizowane sterowanie akcjami obiektu ... 210
Współdzielenie funkcji zdarzeniowych ... 212
Podsumowanie ... 214

Rozdział 4. Kompilacja i techniki optymalizacyjne ... 215
Od C++ do modułu wykonywalnego ... 215
Przyspieszanie kompilacji ... 217
Prekompilowane nagłówki ... 218
Inne metody przyspieszania kompilacji ... 219
Rozszerzenia kompilatora i konsolidatora w wersji 5. C++Buildera ... 221
Kompilacja „w tle” ... 222
Pozostałe nowości kompilatora ... 222
Nowe funkcje konsolidatora ... 223
Podstawowe zasady optymalizacji ... 224
Optymalizacja szybkości wykonania aplikacji ... 226
Przykład optymalizacji — konstruktor krzyżówek ... 228
Opcje projektu wpływające na szybkość generowanego kodu ... 231
Wykrywanie „wąskich gardeł” aplikacji ... 233
Optymalizacja założeń projektowych i algorytmów ... 237
Wysokopoziomowe techniki optymalizowania generowanego kodu ... 245
Techniki optymalizacji danych ... 258
Programowanie na poziomie asemblera ... 261
Optymalizacja uwarunkowań zewnętrznych ... 266
Optymalizacja szybkości — wnioski końcowe ... 266
Optymalizacja innych aspektów aplikacji ... 267
Optymalizacja rozmiaru modułu wynikowego ... 267
Optymalizacja innych czynników ... 268
Podsumowanie ... 269

Rozdział 5. Uruchamianie, śledzenie przebiegu i testowanie aplikacji ... 271
Projektowe uwarunkowania śledzenia aplikacji ... 273
Programistyczne uwarunkowania śledzenia aplikacji ... 275
Podstawowe techniki usuwania błędów aplikacji ... 275
Wyprowadzanie informacji testowych ... 277
Wykorzystanie asercji ... 281
Globalna obsługa wyjątków ... 283
Specyficzne uwarunkowania semantyczne ... 284
Zintegrowany debugger C++Buildera ... 285
Zaawansowane wykorzystanie punktów przerwań ... 286
Nowości C++Buildera 5 związane z punktami przerwań ... 289

Okna zintegrowanego debuggera 289
Podgląd i modyfikacja wyrażeń testowych 293
Inspektor śledzenia 294
CodeGuard 296
Włączanie do aplikacji i konfigurowanie CodeGuarda 296
Wykorzystanie CodeGuarda 298
Zaawansowane techniki śledzenia 305
Znajdowanie przyczyny ogólnego błędu ochrony 305
Podłączanie się do uruchomionego procesu 306
Debugger systemowy 307
Śledzenie zdalne 308
Śledzenie bibliotek DLL 309
Inne narzędzia śledzenia 310
Testowanie aplikacji 311
Etapy i techniki testowania 312
Podsumowanie 313

Rozdział 6. Komponenty biblioteki VCL 315

Wprowadzenie do biblioteki VCL 316
Klasa TObject — początek przygody 316
Tworzenie aplikacji przy użyciu istniejących obiektów 318
Jak używać biblioteki VCL 320
Rozszerzenia języka C++ w systemie C++Builder 5 322
Zapis i odczyt danych ze strumieni 329
Zapis i odczyt ze strumienia obiektów złożonych 330
Zapis i odczyt ze strumienia właściwości niepublikowanych 331
Udoskonalenia standardowych elementów sterujących 335
Biblioteka dynamiczna COMCTL32 335
Rozszerzenia standardowych elementów sterujących wprowadzone w systemie C++Builder 337
Rozszerzenia obiektów klasy THeader 339
Obsługa niestandardowych funkcji rysowania w klasie TToolBar 340
Udoskonalenia standardowych elementów sterujących — podsumowanie 340
Inne rozszerzenia biblioteki VCL 341
Nowe możliwości menu i tekstów podpowiedzi 341
Dostęp do rejestru systemowego 341
Nowa dokumentacja biblioteki VCL 342
Nowy komponent — TApplicationEvents 342
Rozszerzenia klasy TIcon 343
Inne rozszerzenia biblioteki VCL 343
Poprawianie biblioteki VCL, czyli trochę więcej niż klasa TStringList 343
Klasa TStringList 343
Jak przechowywać obiekty nie należące do biblioteki VCL? 344
Łączenie łańcuchów z obiektami tego samego typu 345

Tworzenie łańcucha zdarzeń ... 354
Sortowanie list ... 356
Kilka poprawek ... 356
Zdarzenia związane z zaawansowaną obsługą niestandardowych funkcji rysowania ... 358
Komponent TTreeView ... 358
Komponent TListView ... 359
Komponent TToolBar ... 359
Przykłady wykorzystania zdarzeń związanych z zaawansowaną obsługą niestandardowych funkcji rysowania ... 359
Kreator apletów Panelu sterowania ... 359
Podstawy działania apletu ... 360
Komponenty firm niezależnych ... 369
Wady i zalety komponentów firm niezależnych ... 369
Gdzie szukać komponentów dla C++Buildera? ... 370
Podsumowanie ... 371

Rozdział 7. Tworzenie własnych komponentów ... 373

Podstawy tworzenia komponentów ... 373
Rozszerzanie możliwości klasy bazowej ... 374
Założenia projektowe ... 377
Tworzenie komponentów niewidocznych ... 377
Właściwości ... 377
Zdarzenia ... 386
Metody ... 389
Definiowanie wyjątków związanych z komponentem ... 391
Przestrzenie nazw — dyrektywa namespace ... 393
Obsługa komunikatów ... 395
Etap projektowania a etap wykonania ... 397
Powiązania między komponentami ... 399
Projektowanie komponentów wizualnych ... 403
TCanvas ... 403
Wykorzystanie kontrolek graficznych ... 405
Reagowanie na zdarzenia pochodzące od myszy ... 407
Przykład zastosowania ... 409
Rozbudowa kontrolek okienkowych ... 417
Tworzenie własnych kontrolek bazodanowych ... 430
Połączenie kontrolki z bazą danych ... 430
Aktualizowanie zawartości kontrolki — zdarzenie OnDataChange ... 433
Zapis zmian do bazy — zdarzenie OnUpdateData ... 434
Komunikat CM_GETDATALINK ... 437
Rejestracja komponentów ... 439
Podsumowanie ... 441

Rozdział 8. Edytory komponentów i edytory właściwości ... 443
Tworzenie edytorów właściwości ... 444
Metoda GetAttributes() ... 454
Metoda GetValue() ... 455
Metoda SetValue() ... 456
Metoda Edit() ... 457
Metoda GetValues() ... 461
Właściwości klasy TPropertyEditor ... 462
Właściwości i wyjątki ... 462
Rejestracja edytora właściwości ... 464
Uzyskiwanie informacji o typie właściwości spoza biblioteki VCL ... 466
Samodzielne tworzenie informacji o typie właściwości spoza biblioteki VCL ... 472
Zasady zastępowania edytorów właściwości ... 474
Wykorzystanie grafiki w edytorach właściwości ... 474
Metoda ListMeasureWidth() ... 478
Metoda ListMeasureHeight() ... 479
Metoda ListDrawValue() ... 479
Metoda PropDrawValue() ... 484
Metoda PropDrawName() ... 485
Instalowanie pakietów zawierających edytory ... 487
Wykorzystanie kolekcji obrazków w edytorach właściwości ... 488
Metoda GetAttributes() ... 493
Metoda GetComponentImageList() ... 494
Metoda GetValues() ... 494
Metody ListMeasureWidth() i ListMeasureHeight() ... 495
Metoda ListDrawValue() ... 496
Metoda PropDrawValue() ... 498
Wykorzystanie listy obrazków komponentu rodzicielskiego ... 500
Uniwersalne podejście do edycji właściwości ImageIndex ... 504
Tworzenie edytorów komponentów ... 508
Metoda Edit() ... 513
Metoda EditProperty() ... 517
Metoda GetVerbCount() ... 519
Metoda GetVerb() ... 519
Metoda PrepareItem() ... 520
Metoda ExecuteVerb() ... 524
Metoda Copy() ... 525
Rejestracja edytorów komponentów ... 527
Wykorzystanie predefiniowanych obrazków w edytorach komponentów i edytorach właściwości ... 527
Dodawanie plików zasobowych do pakietów ... 528
Wykorzystanie zasobów w edytorach komponentów i edytorach właściwości ... 529

Podział właściwości na kategorie i ich rejestracja ... 532
Kategorie i ich tworzenie ... 533
Zaliczenie właściwości do konkretnej kategorii ... 535
Podsumowanie ... 541

Rozdział 9. Programowanie zagadnień telekomunikacyjnych ... 543
Komunikacja szeregowa ... 543
Protokoły komunikacyjne ... 543
Protokoły jako maszyny z pamięcią stanu ... 547
Efektywność a niezawodność ... 548
Architektura aplikacji ... 548
Protokoły internetowe — SMTP, FTP, HTTP i POP3 ... 551
Wycieczka po Palecie Komponentów ... 551
Serwer pogawędki ... 552
Klient pogawędki ... 556
Klient poczty elektronicznej ... 560
Serwer HTTP ... 567
Klient FTP ... 570
Podsumowanie ... 576

Rozdział 10. Programowanie serwerów WWW ... 577
Moduły WWW ... 577
Web Serwer Application Wizard ... 578
CGI ... 578
WinCGI ... 578
ISAPI/NSAPI ... 578
CGI czy ISAPI? ... 579
Podstawowe komponenty WebBrokera ... 579
TWebDispatcher ... 580
TWebModule ... 580
TWebResponse ... 581
TWebRequest ... 582
Serwery WWW ... 583
Komponenty-producenci WebBrokera ... 586
TPageProducer ... 587
TDataSetPageProducer ... 589
TDataSetTableProducer ... 592
TQueryTableProducer ... 594
Zarządzanie stanem sesji ... 597
„Gruby URL” ... 597
Cookies ... 597
Ukryte pola formularzy HTML ... 598
Bezpieczeństwo aplikacji sieciowych ... 600
SSL — warstwa bezpiecznych gniazd ... 601
Autoryzacja ... 601

Nagłówki HTTP ... 602
Problem z biblioteką VCL ... 603
Zabezpieczanie aplikacji WWW ... 603
Solidność zabezpieczeń ... 604
Kryptografia to jest to ... 604
HTML i XML ... 608
XML ... 608
InternetExpress ... 609
Przykład — kartoteka zamówień ... 610
Podsumowanie ... 615

Rozdział 11. Biblioteki DLL ... 617

Kreator DLL Wizard ... 617
Tworzenie i wykorzystywanie bibliotek DLL ... 618
Łączenie statyczne ... 619
Dynamiczne importowanie funkcji z biblioteki DLL ... 621
Eksportowanie klas z biblioteki DLL ... 625
Biblioteki DLL a pakiety ... 627
Formularze SDI w bibliotekach DLL ... 629
Formularze potomne MDI w bibliotekach DLL i pakietach ... 630
Wykorzystanie w C++Builderze bibliotek stworzonych w Visual C++ ... 638
Wykorzystanie bibliotek C++Buildera w aplikacjach Visual C++ ... 639
„Wtyczki" DLL ... 640
Przykład — menedżer wtyczek TBCB5PluginManager ... 644
Końcowe uwagi na temat implementacji wtyczek ... 645
Podsumowanie ... 645

Rozdział 12. Programowanie COM ... 647

Serwery i klienci COM ... 648
Interfejsy wychodzące i ujścia zdarzeń ... 648
Tworzenie serwera COM ... 649
Wybór typu serwera ... 649
Wybór modelu wątkowego ... 650
Utworzenie serwera ... 651
Dodanie obiektu COM ... 652
Przegląd wygenerowanego kodu ... 655
Uzupełnianie treści metod ... 659
Ulepszenie obsługi błędów ... 661
Implementacja metody generującej zdarzenie ... 663
Implementacja „klasycznego" interfejsu ... 664
Generowanie zdarzeń serwera ... 667
Tworzenie DLL dla proxy i stuba ... 672
Tworzenie klienta COM ... 678
Importowanie biblioteki typu ... 679
Przegląd wygenerowanych konstrukcji C++ ... 680

Tworzenie i wykorzystywanie obiektu COM serwera ... 684
Przechwytywanie zdarzeń zbudowanych na podstawie interfejsów dyspozycyjnych . 685
Operowanie „klasycznymi” interfejsami ... 688
Tworzenie ujścia zdarzeń zbudowanego na klasycznym interfejsie ... 689
Literatura zalecana ... 693
Podsumowanie ... 694
Rozdział 13. Programowanie rozproszone — DCOM ... 695
Czym jest DCOM? ... 695
Rodzina systemów Windows a DCOM ... 696
Konfigurowanie DCOM — program DCOMCnfg ... 696
Ustawienia ogólnosystemowe ... 697
Ustawienia związane z konkretnym serwerem ... 700
Zastosowanie DCOM — przykład ... 702
Tworzenie aplikacji serwera ... 702
Tworzenie aplikacji klienta ... 704
Konfigurowanie uprawnień dostępu i uruchamiania serwera ... 706
Konfigurowanie identyfikacji ... 707
Uruchomienie przykładu ... 708
Bezpieczeństwo programowane ... 708
Podsumowanie ... 723
Rozdział 14. Interfejs Win32 i jego wykorzystanie ... 725
Oprogramowanie pośrednie a funkcje Win32 ... 725
Krótka historia Windows i jego API ... 726
Kategorie funkcji interfejsu Win32 ... 730
Usługi systemowe ... 734
Interfejs GDI ... 736
Obsługa multimediów ... 737
Standardowe elementy interfejsu użytkownika ... 739
Elementy i funkcje powłoki systemu ... 742
Obsługa ustawień regionalnych ... 743
Usługi sieciowe ... 743
Budowa i działanie aplikacji dla Windows ... 743
Funkcja WinMain() ... 744
Uchwyty okien ... 746
Komunikaty ... 746
Praktyczne przykłady użycia funkcji Win32 ... 749
Uruchomienie programu z innego programu ... 750
Podstawy obsługi plikowych operacji wejścia-wyjścia ... 753
Magiczne funkcje powłoki ... 764
Obsługa multimediów ... 774
Identyfikatory GUID i ich wykorzystanie ... 778
Pobieranie informacji o systemie ... 779
Odczytanie nazwy użytkownika ... 779

Odczytanie nazwy komputera ... 780
Ustalenie lokalizacji plików tymczasowych ... 781
Zarządzanie stacją roboczą ... 793
Animacja okien ... 795
Kółko graniaste, czyli okna dowolnych kształtów ... 797
Tworzenie apletów Panelu sterowania — metoda tradycyjna ... 806
Podsumowanie ... 816

Rozdział 15. Techniki multimedialne ... 819

Interfejs GDI ... 820
Interfejs programowy Windows i konteksty urządzeń ... 820
Kontekst urządzenia a VCL, czyli klasa TCanvas ... 821
Zmiana ustawień rysowania ... 824
Przykład — zegar analogowy ... 826
Wyświetlanie grafiki rastrowej ... 827
Mapy bitowe w Windows ... 827
Klasa TBitmap ... 828
Mapy bitowe w formacie JPEG ... 829
Mapy bitowe w formacie GIF ... 830
Mapy bitowe w formacie PNG ... 830
Przetwarzanie obrazu ... 833
Odczytywanie i wyświetlanie parametrów obrazu ... 834
Dostęp do pikseli poprzez właściwość TCanvas->Pixels ... 835
Tworzenie obrazów ... 836
Dostęp do pikseli poprzez właściwość ScanLine ... 837
Przekształcenia punktowe — dyskryminacja i konwersja koloru na odcienie szarości ... 838
Przekształcenia globalne — wyrównanie histogramu ... 841
Przekształcenia geometryczne — powiększenie ... 843
Przekształcenia splotowe — wygładzanie i detekcja krawędzi ... 846
Odtwarzanie zapisów audio, wideo i płyt CD ... 848
Interfejs MCI ... 848
Odtwarzanie spróbkowanych zapisów dźwięku ... 855
Uwagi końcowe ... 862
Podsumowanie ... 862

Rozdział 16. Zaawansowane techniki graficzne — OpenGL i DirectX ... 863

Wprowadzenie do standardu OpenGL ... 863
OpenGL a Direct3D ... 864
Struktura polecenia OpenGL ... 864
Aktualizacja zawartości sceny w funkcji OnIdle() ... 865
Wykorzystanie funkcji OpenGL ... 865
Etap 1 — inicjalizacja podsystemu OpenGL ... 866
Etap 2 — ustalenie parametrów oświetlenia i cieniowania ... 872
Etap 3 — przekształcenia 3D ... 874
Etap 4 — rysowanie obiektów pierwotnych ... 877

Etap 5 — wymiana buforów ... 885
Przykładowy program wykorzystujący funkcje OpenGL ... 886
Podsumowanie ... 887
Materiały uzupełniające ... 887
Wprowadzenie do standardu DirectX ... 888
DirectX a model COM ... 888
Nieobiektowe funkcje DirectX ... 889
Interfejs DirectDraw ... 889
Inicjalizacja obiektu DirectDraw ... 889
Definiowanie ustawień ekranu ... 891
Powierzchnie ... 892
Rysowanie na powierzchniach DirectDraw za pomocą funkcji GDI ... 894
Wyświetlanie map bitowych na powierzchniach DirectDraw ... 896
Przykładowy program wykorzystujący funkcje DirectDraw ... 900
Podsumowanie ... 900
Interfejs DirectSound ... 901
Inicjalizacja obiektu DirectSound ... 901
Utworzenie bufora pomocniczego ... 902
Program przykładowy — równoległe odtwarzanie kilku dźwięków ... 908
Pozostałe elementy standardu DirectX ... 909
Materiały uzupełniające ... 909
Podsumowanie ... 909

Rozdział 17. Tworzenie dokumentacji i plików pomocy ... 911

Dziesięć przykazań autora dokumentacji ... 912
Rodzaje dokumentacji ... 913
Metodologia tworzenia dokumentacji elektronicznej ... 914
Kategorie systemów pomocy ... 914
Formaty plików pomocy ... 917
WinHelp — sprawdzony standard ... 919
Narzędzia do tworzenia plików pomocy ... 921
Kontekstowość systemu pomocy ... 921
Program Microsoft Help Workshop ... 923
Definiowanie lokalnych okienek pomocy ... 929
Dodatkowe możliwości kompilatora Help Workshop ... 929
Microsoft HTML Help ... 931
Obsługa pomocy w bibliotece VCL ... 932
Właściwości ... 932
Metody ... 933
Zdarzenia ... 936
Materiały i narzędzia dla twórców dokumentacji ... 936
Publikacje książkowe ... 936
Narzędzia do projektowania systemów pomocy ... 937
Podsumowanie ... 941

Rozdział 18. Instalowanie i aktualizowanie oprogramowania 943
Instalacja i dezinstalacja oprogramowania 943
Narzędzia do tworzenia programów instalacyjnych 943
Program Install Maker 944
Pliki CAB i INF 948
Pliki CAB 949
Pliki INF 951
Internetowe pakiety instalacyjne 956
Wersje, uaktualnienia i poprawki 958
Kontrola wersji 958
Uaktualnienia 960
Poprawki 962
Program Patch Maker 963
Wskazówki odnośnie uaktualnień i poprawek 965
System kontroli wersji TeamSource 965
Dla kogo przeznaczony jest TeamSource? 966
Dlaczego używać TeamSource? 966
Kiedy używać TeamSource? 966
Gdzie można używać TeamSource? 967
Jak używać TeamSource? 967
Widoki projektu w programie TeamSource 973
Mechanizmy kontroli wersji 977
Zakładki 977
Blokady 978
Program InstallShield Express 979
Instalacja programu InstallShield 979
Pierwsze kroki 980
Testowanie programu instalacyjnego 987
Podsumowanie 988

Skorowidz 989

Rozdział 4. Kompilacja i techniki optymalizacyjne

Rozdział ten poświęcony jest tym elementom C++Buildera, które dokonują przekształcenia źródłowej postaci projektu w końcowe pliki wykonywalne — czyli kompilatorowi i konsolidatorowi — a dokładniej dwóm najważniejszym aspektom ich pracy: optymalności samego zabiegu owego „przekształcania" oraz optymalności *produktu końcowego*, głównie pod względem szybkości jego wykonywania (chociaż także i innych czynników, jak na przykład zajętość pamięci czy efektywność operacji dyskowych).

Większość obecnych kompilatorów, w tym również oczywiście C++Builder, dokonuje różnego rodzaju optymalizacji tworzonego przekładu, by uczynić go możliwie szybkim i zwięzłym. Czynności kompilacji i optymalizacji nierozerwalnie splatają się ze sobą, jeżeli pod pierwszym z tych określeń rozumieć tłumaczenie fragmentów kodu w języku wysokiego poziomu (C++) na równoważne fragmenty w kodzie maszynowym, zaś pod drugim — możliwie najlepszy dobór instrukcji tego kodu pod względem jego efektywności i rozmiaru.

Niezależnie jednak od najbardziej spektakularnych przejawów „inteligencji" kompilatora największy przyczynek do efektywności kodu wynikowego wnieść może sam programista, w przeciwieństwie bowiem do kompilatora, decydującego o tym, *jak* przetłumaczyć ustalony fragment kodu źródłowego, określa on, *co* ma zostać przetłumaczone; najbardziej nawet wyrafinowany kompilator, nie znający istoty rozwiązywanego problemu i intencji programisty, nie jest w stanie zniwelować skutków nieoptymalnego kodowania. W tym rozdziale omawiamy więc nie tylko różnorodne techniki optymalizacji automatycznej, lecz przede wszystkim zwracamy uwagę na te aspekty konstruowania aplikacji, które w największym stopniu odpowiedzialne są za efektywność wygenerowanego kodu.

Od C++ do modułu wykonywalnego

Co tak naprawdę dzieje się, gdy naciskamy klawisz *F9*, bądź wybieramy z menu głównego którąkolwiek opcję powodującą skompilowanie projektu? Co prawda odpowiedź na to pytanie nie jest bynajmniej niezbędna dla samego tworzenia (i kompilowania)

projektów, należy jednak zdawać sobie sprawę z oczywistego faktu, iż kompilator jest na tyle istotnym elementem C++Buildera, iż bez niego pozostałe elementy stałyby się niemal bezużyteczne! Kompilator ten stosuje daleko posunięte zabiegi optymalizacyjne, dając w efekcie bardzo efektywny kod o jakości konkurencyjnej w stosunku do wytworów innych współczesnych kompilatorów C++. W procesie przekształcania kodu źródłowego w C++ w plik wykonywalny **.exe* lub bibliotekę **.dll* wyodrębnić można cztery następujące fazy:

- *przetwarzanie wstępne* (*preprocessing*) — w tej fazie następuje rozwijanie makr i dyrektyw zawartych w kodzie źródłowym; w szczególności włączane są pliki nagłówkowe określone przez dyrektywy `#include`;
- *rozbiór syntaktyczny i semantyczny* — w kodzie stanowiącym rezultat pracy preprocesora wydzielane są poszczególne jednostki składniowe i następuje określenie znaczenia (semantyki) tych jednostek; w ostateczności generowane jest tzw. drzewo wyprowadzenia syntaktycznego (ang. *syntax tree*), stanowiące podstawę generowania kodu wynikowego;
- *generowanie kodu* — poszczególne instrukcje zastępowane są odpowiednikami w kodzie maszynowym; generowany kod jest optymalizowany głównie pod kątem tzw. parowania instrukcji (ang. *instruction pairing*) i cech specyficznych dla danego procesora. Kod wynikowy każdego z modułów zapisywany jest w odrębnym pliku **.obj*; na końcu każdego z tych plików dołączana jest opcjonalnie informacja niezbędna w procesie śledzenia symbolicznego (*debugging*), odzwierciedlająca powiązanie poszczególnych fragmentów wygenerowanego kodu binarnego z poszczególnymi instrukcjami i elementami danych kodu źródłowego;
- *konsolidacja* — konsolidator analizuje zawartość każdego z wynikowych plików **.obj*, tworząc globalną tablicę symboli, na podstawie której realizowane są następnie międzymodułowe odwołania do funkcji i danych definiowanych w poszczególnych modułach. Kod zawarty w plikach **.obj* łączony jest z kodem plików zasobowych i statycznie dołączanych bibliotek, dając w efekcie gotowy do wykonania plik **.exe* lub bibliotekę **.dll*.

Celowo napisaliśmy „fazy", a nie „etapy" — bowiem wymienienie powyższych faz w określonej kolejności nie ma bynajmniej sugerować kolejnych, odrębnych stadiów obróbki kodu źródłowego. Kompilator, analizując kod źródłowy, posługuje się metodą tzw. zejścia rekurencyjnego z nieograniczonym wyprzedzeniem przeglądania (ang. *recursive descent model with infinite lookahead*), co oznacza, iż każda jednostka semantyczna rozpatrywana jest w podziale na prostsze jednostki składowe, zaś do rozpoznania kolejnej jednostki wymagane jest wczytanie pewnej, nie ograniczanej z góry, liczby znaków kodu[1]. Rekursywny charakter analizy polega natomiast na równoległym, rekursywnym wywoływaniu procedur realizujących wymienione fazy.

Każda z wymienionych faz stwarza pewną okazję do dokonywania optymalizacji, niemniej jednak wszelkie czynności optymalizacyjne podzielić można na dwie kategorie: te mające wpływ na postać drzewa syntaktycznego, odnoszące się więc do kodu źródłowego

[1] W przeciwieństwie do przeglądania z wyprzedzeniem o jeden znak (ang. *one-char lookahead* lub *near lookahead*), kiedy to do zidentyfikowania jednostki syntaktycznej wystarcza jej początkowy znak — *przyp. tłum.*

i zwane stąd *wysokopoziomowymi,* i te *niskopoziomowe* związane ściśle z architekturą docelowego procesora i repertuarem jego instrukcji. Do pierwszej z wymienionych kategorii zaliczyć można m.in.: upraszczanie podwyrażeń (ang. *subexpression folding*), polegające na zastępowaniu działań na wartościach stałych ich wynikami, zastępowanie mnożeń i dzieleń przez potęgi dwójki operacjami przesunięć bitowych, rozwijanie funkcji wstawialnych (*inlining*) itp. Optymalizacje niskopoziomowe mają charakter bardziej subtelny, a ich przykładem może być eliminacja sąsiadujących rozkazów nie powodujących w sumie żadnego efektu, jak np. dwa początkowe rozkazy w poniższej sekwencji:

```
push eax
pop  eax
mov  ebx,eax
push edx
```

czy też eliminacja zbędnych rozkazów, jak w poniższym przykładzie:

```
mov  edx,A
mov  eax,B
add  eax,edx
push eax
mov  edx,A   ←
mov  eax,F
sub  eax,edx
push eax
```

gdzie rozkaz wskazany przez strzałkę jest po prostu zbędny.

Nie są natomiast wykonywane żadne optymalizacje, wynikające z zależności przekraczających granice poszczególnych funkcji.

Większość opcji sterujących przebiegiem kompilacji i postacią kodu wynikowego dostępna jest na kartach: *Compiler*, *Advanced Compiler*, *Linker* i *Advanced Linker* opcji projektu, niektóre dostępne są jednak tylko z poziomu głównego pliku projektu (**.bpr*) lub tylko w wywołaniach kompilatora z wiersza poleceń.

Jedną z nowości wersji 5 C++Buildera jest to, iż plik **.bpr* ma format charakterystyczny dla języka XML. Jego zawartość edytować można z poziomu IDE, wybierając opcję *Project|Edit Option Source* z menu głównego. Opcje dotyczące (odpowiednio): kompilatora, konsolidatora, kompilatora zasobów, kompilatora Object Pascala i Asemblera znajdują się w sekcji `<OPTIONS>` w pozycjach (odpowiednio): `<CFLAG1 … >`, `<LFLAGS … >`, `<RFLAGS … >`, `<PFLAGS … >` i `<AFLAGS … >`.

W podkatalogu *Examples* lokalnej instalacji C++Buildera znajduje się ciekawy projekt o nazwie *WinTools*. Ilustruje on znaczenie poszczególnych opcji kompilatora i innych programów narzędziowych C++Buildera, których kompletny wykaz (wraz z opisem poszczególnych opcji) znaleźć można w systemie pomocy.

Przyspieszanie kompilacji

Kompilator C++Buildera, niezależnie od wysokiej jakości tworzonego kodu, sam w sobie jest szybkim programem. Jest niemal dwa razy szybszy od kompilatora GNU C++ i porównywalny pod względem szybkości z kompilatorem Visual C++. Dla użytkowni-

ków posługujących się jednocześnie Delphi wydaje się on jednak cokolwiek powolny, jeżeli rozpatrywać porównywalne pod względem rozmiaru aplikacje w C++ i Object Pascalu; porównując jednak Delphi z C++Builderem należy mieć na uwadze różnorodne czynniki mające wpływ na tę różnicę szybkości, między innymi:

- C++ wykorzystuje intensywnie pliki nagłówkowe, których brak jest w Object Pascalu[2]. Ze względu na to, iż pliki te mogą być zagnieżdżane, rzeczywisty kod, z którym uporać się musi kompilator, przybrać może rozmiary znacznie większe od tych wynikających z pierwszego spojrzenia na dany projekt — skomplikowane, rekurencyjne zagnieżdżenie kilku zaledwie plików nagłówkowych o długości 10 wierszy każdy może dać w efekcie kod o długości setek tysięcy wierszy! Z oczywistych względów musi się to kompilować dłużej niż 10 – 20-wierszowy projekt w Delphi.
- W Object Pascalu nie występują makra, które w C++ interpretowane są w czasie kompilacji.
- Charakterystyczny dla C++, nieobecny w Object Pascalu, mechanizm szablonów znacznie komplikuje proces analizy kodu źródłowego.
- Semantyka C++ podporządkowana jest standardom ANSI i jest nieporównanie bardziej złożona od „gramatyki" Object Pascala, stanowiącej arbitralny standard Borlanda.

Generalnie C++ oferuje programiście o wiele większe możliwości w zakresie tworzenia aplikacji niż Delphi oparte na Object Pascalu. Za ofertę tę trzeba jednak zapłacić cenę w postaci bardziej czasochłonnej kompilacji i (często) mniej czytelnego kodu źródłowego. Tym większego znaczenia nabierają więc oferowane przez C++Builder mechanizmy umożliwiające przyspieszenie kompilacji — omówimy je teraz w kolejnych punktach.

Prekompilowane nagłówki

Jedną z najbardziej skutecznych metod przyspieszania kompilacji jest unikanie powtórnej kompilacji tych samych plików nagłówkowych lub ich powtarzającej się sekwencji w różnych modułach źródłowych. Zaznaczając opcję *Use pre-compiled headers* w sekcji *Pre-compiled headers* na karcie *Compiler* opcji projektu spowodujemy zapisywanie we wskazanym pliku dyskowym skompilowanej postaci każdego z nagłówków, tworzonej przy pierwszym napotkaniu odwołania do tegoż nagłówka i wykorzystywanej przy kolejnych odwołaniach. Zaznaczając opcję *Cache pre-compiled headers*, zyskujemy dalsze przyspieszenie kompilacji poprzez buforowanie prekompilowanych nagłówków w pamięci operacyjnej.

Napotkanie przez kompilator (w module źródłowym) dyrektywy `#pragma hdrstop` stanowi dla niego polecenie zaprzestania wykorzystywania prekompilowanych nagłówków podczas dalszej kompilacji modułu. Nagłówki włączane do tegoż modułu przez dyrek-

[2] Co prawda w Pascalu również można „wstawiać" do kodu pliki źródłowe — za pomocą dyrektywy `{$I` — rzadko jednak wstawianie to jest zagnieżdżane, tym bardziej rekurencyjnie; „dołączanie" do modułu stosownych definicji odbywa się zazwyczaj przy użyciu dyrektyw `uses`. Nie sposób więc porównywać dyrektyw `{$I` z dyrektywami `#include` — *przyp. tłum.*

tywy `#include` poprzedzające dyrektywę `#pragma hdrstop` podlegać będą prekompilacji, należy jednak wspomnieć tutaj o dość istotnym uwarunkowaniu tego mechanizmu. Otóż prekompilacji podlegają nie tyle oddzielne nagłówki, ile ich konkretne *zestawy* w konkretnej *kolejności* (wynikającej z kolejności odnośnych dyrektyw `#include`). Dwa różne moduły źródłowe współdzielić więc będą tę samą prekompilowaną porcję nagłówków jedynie wówczas, gdy lista dołączanych plików nagłówkowych (przed dyrektywą `#pragma hdrstop`) będzie w obydwu tych modułach *identyczna co do zestawu i kolejności* plików. W modułach generowanych automatycznie przez C++Builder taką prekompilowaną listę stanowią nagłówki charakterystyczne dla biblioteki VCL — lista ta poprzedza dyrektywę `#pragma hdrstop`, po której odbywa się dołączanie nagłówków charakterystycznych dla danego modułu i nie podlegających prekompilacji. Oto ilustracja tej idei na przykładzie dwóch (fikcyjnych) modułów źródłowych:

Wydruk 4.1.
Dwa moduły źródłowe współdzielące prekompilowaną listę nagłówków

```
//------------------------------------   //------------------------------------
// LoadPage.cpp                          // ViewOptions.cpp

#include <vcl.h>                         #include <vcl.h>
#include <System.hpp>                    #include <System.hpp>
#include <Windows.hpp>                   #include <Windows.hpp>
#include "SearchMain.h"                  #include "SearchMain.h"
#pragma hdrstop                          #pragma hdrstop

#include "LoadPage.h"                    #include <Graphics.hpp>
#include "CacheClass.h"                  #include "ViewOptions.h"
//------------------------------------   //------------------------------------

// Kod...                                // Kod...
```

Jak pokazuje praktyka, umiejętne grupowanie dołączanych plików nagłówkowych skutkować może nawet *dziesięciokrotnym* przyspieszeniem kompilacji!

Autorzy oryginału proponują w tym miejscu artykuł zawierający więcej informacji na temat prekompilowanych nagłówków, znajdujący się pod adresem: *http://www.bcbdev.com/articles/pch.htm*.

Inne metody przyspieszania kompilacji

Najbardziej oczywistym sposobem skracania czasu kompilacji projektu jest unikanie kompilowania tych fragmentów kodu, które i tak nie zostaną w projekcie wykorzystane. Dotyczy to w pierwszym rzędzie zbędnych plików nagłówkowych — związane z nimi dyrektywy `#include` najprościej po prostu „wykomentować".

Istnieje jednak istotny wyjątek od tej zasady. Jak przed chwilą napisaliśmy, prekompilacja nagłówków przynosi widoczne korzyści jedynie wówczas, gdy dwa moduły (lub większa ich liczba) posługują się identyczną listą dołączanych plików nagłówkowych. W poniższym przykładzie dwa (fikcyjne) moduły źródłowe nie spełniają tego warunku:

Wydruk 4.2.
Sytuacja, w której dołączenie niewykorzystywanych plików nagłówkowych przyspieszy kompilację

```
//---------------------------------
// FirstModule.cpp
#include <vcl.h>
#include <System.hpp>
#include <Windows.hpp>

#include "ScanModules.h"
#pragma hdrstop
#include "LoadPage.h"
#include "CacheClass.h"
//---------------------------------
// Kod ...
```

```
//---------------------------------
// SecondModule.cpp
#include <vcl.h>
#include <System.hpp>
#include <Windows.hpp>
#include "SearchMain.h"

#pragma hdrstop
#include <Graphics.hpp>
#include "ViewOptions.h"
//---------------------------------
// Kod ...
```

Jeżeli jednak do modułu `FirstModule` wstawić (w odpowiednim miejscu) dyrektywę `#include "SearchMain.h"`, zaś do modułu `SecondModule` — dyrektywę `#include "ScanModules.h"`, moduły zaczną kompilować się szybciej, bowiem zysk wynikający ze współdzielenia prekompilowanej listy nagłówków przewyższy z pewnością stratę czasu spowodowaną kompilacją niewykorzystywanych plików nagłówkowych.

Równie oczywistym sposobem zaoszczędzenia czasu kompilacji jest kompilowanie tylko tych modułów źródłowych, które faktycznie tej kompilacji wymagają. Wybierając opcję *Make...* z menu *Project* nakazujemy kompilatorowi skompilować tylko te moduły, które jeszcze kompilowane nie były (tj. nie posiadają odpowiadającego pliku **.obj*) oraz te, których treść była modyfikowana od czasu ostatniej kompilacji; wybranie opcji *Build...* spowodowałoby natomiast kompilację wszystkich modułów projektu, trwającą zazwyczaj nieco dłużej.

Kompilacja w trybie *Make* nie zawsze jednak daje pożądane rezultaty. Jedynym bowiem kryterium, którym kieruje się kompilator, oceniając konieczność ponownej kompilacji modułu, jest data jego ostatniej modyfikacji (w konfrontacji z datą ostatniej modyfikacji odpowiedniego pliku **.obj*); tymczasem modyfikacja kodu źródłowego modułu nie jest bynajmniej *jedyną* okolicznością uzasadniającą jego rekompilację — równie istotną przesłanką może być np. zmodyfikowanie opcji projektu, której to przesłanki kompilator nie weźmie jednak pod uwagę i gwarancję uzyskania aktualnego kodu wynikowego daje wówczas tylko kompilacja w trybie *Build*.

Kolejne źródło oszczędności czasu kompilacji kryje się w szczegółowości generowanego kodu, a konkretnie — w informacjach symbolicznych dla debuggerów. W sytuacji, gdy testowanie programu polega wyłącznie na obserwacji zewnętrznych przejawów jego działania (bez pracy krokowej), korzystne może okazać się wyłączenie opcji związanych ze śledzeniem na kartach *Compiler*, *Linker* i *Pascal* opcji projektu; ułatwia to znakomicie przycisk *Release*, znajdujący się na karcie *Compiler* — przywrócenie kompletu opcji stosownych dla trybu śledzenia następuje po kliknięciu przycisku *Full Debug*.

Mechanizmem umożliwiającym przyspieszenie pracy konsolidatora jest tzw. *konsolidacja przyrostowa* (ang. *incremental linking*), polegająca na wykorzystywaniu aktualnych jeszcze informacji pochodzących z wcześniejszych konsolidacji danego projektu zapisywanych w tzw. plikach stanu konsolidacji (ang. *linker state files*). O wykorzystywaniu

tego mechanizmu decyduje opcja *Don't generate state files* na karcie *Linker* opcji projektu — jej zaznaczenie powoduje rezygnację z konsolidacji przyrostowej.

Do skrócenia czasu konsolidacji przyczynia się również rezygnacja z dołączania niepotrzebnych bibliotek — i tak na przykład w aplikacji nie korzystającej z arytmetyki zmiennoprzecinkowej należy zaznaczyć opcję *None* w sekcji *Floating Point* na karcie *Advanced Compiler* (jeżeli dokonamy tego w aplikacji wykonującej operacje zmiennoprzecinkowe, spowodujemy błąd konsolidacji).

Możliwości przyspieszania kompilacji i konsolidacji nie kończą się bynajmniej na samym projekcie — równie istotne może być środowisko sprzętowo-programowe, w którym uruchamiany jest C++Builder. Jego wymagania pod względem pamięci RAM, szybkości procesora i transferu dyskowego plasują się (co tu kryć) powyżej przeciętnej, podobnie zresztą jak w przypadku innych narzędzi typu RAD. Nawet jednak w konkretnej konfiguracji sprzętowej można dokonać kilku prostych zabiegów bezinwestycyjnych, zdolnych poprawić efektywność pracy C++Buildera — w postaci np. zamknięcia zbędnych programów czy defragmentacji dysku.

W komputerach wieloprocesorowych można dokonywać jednoczesnej kompilacji kilku modułów źródłowych. Dostarczany przez Borland program *MAKE* nie umożliwia zrobienia tego w sposób bezpośredni, można jednak uruchomić równolegle kilka jego kopii, każdą dla innego modułu. Można również skorzystać z programu *Make* zawartego w pakiecie *GNU*, uruchamianego z przełącznikiem `-j`; program ten dostępny jest pod adresem *http://sourceware.cygnus.com/cygwin/*, zaś jego dokumentację znaleźć można pod *adresem http://www.gnu.org/software/make*. W wersji 5. C++Buildera nie można co prawda zastosować programu *Make* wprost do pliku projektu — ma on obecnie format XML — nietrudno jednak utworzyć jego odpowiednik w formacie `MAKEFILE` za pomocą opcji *Export Makefile* z menu *Project*.

Wreszcie — nie bez znaczenia jest również sam system operacyjny, a dokładniej jego mobilność: zdaniem autorów systemy Windows NT i Windows 2000 okazują się być bardziej „reaktywnymi" niż Windows 95/98, dostarczając jednocześnie lepszego środowiska dla debuggerów.

Rozszerzenia kompilatora i konsolidatora w wersji 5. C++Buildera

Najbardziej spektakularną spośród nowości kompilatora w wersji 5. C++Buildera jest z pewnością możliwość kompilowania projektu „w tle", równolegle z wykonywaniem innych czynności, toteż przyjrzymy się jej nieco dokładniej. Wśród pozostałych nowości wymienić należy: dodatkowe mechanizmy kompatybilności z Visual C++, nową obsługę plików, bogatszy zestaw przełączników i ostrzeżeń konsolidatora, rozszerzoną informację o błędach itp.

Kompilacja „w tle”

Podczas gdy kompilator wykonuje swą pracę, możliwe jest kontynuowanie pracy nad projektem — przeglądanie i edycja plików, formularzy itp. — co w przypadku złożonego projektu może w znacznym stopniu przyczynić się do zwiększenia produktywności programisty. Rozwiązanie takie stwarza jednak kilka poważnych problemów, z których najistotniejsze są dwa:

- dokonywanie zmian w pliku źródłowym w momencie, gdy plik ten jest odczytywany przez kompilator stanowi poważne zagrożenie dla integralności tego, co tak naprawdę przez kompilator zostanie odczytane;
- nawet jeżeli kompilator i edytor kodu nie wchodzą sobie w drogę, wynik kompilacji uzależniony jest od tego, czy modyfikacja danego pliku źródłowego dokonana została przed jego skompilowaniem, czy też po skompilowaniu; pod znakiem zapytania staje więc nie tylko integralność poszczególnych plików, lecz integralność projektu jako całości.

C++Builder zapewnia rozwiązanie jedynie pierwszego z wymienionych problemów — zanim mianowicie kompilator przystąpi do odczytu danego pliku źródłowego, nadaje mu chwilowo atrybut „tylko do odczytu”, co chronić ma ów plik przed ew. modyfikacjami podczas odczytywania; po skompilowaniu zawartości pliku przywracane są jego poprzednie atrybuty. Nie istnieją natomiast analogiczne mechanizmy chroniące integralność *projektu* jako całości, co stwarza pewną sytuację hazardu — wynik kompilacji zależny jest od względnej szybkości pracy kompilatora i programisty modyfikującego kod źródłowy.

Kompilacja „w tle” wiąże się z innymi jeszcze, mniej poważnymi ograniczeniami — nie jest mianowicie stosowana przy projektach *dzielonych na pakiety* oraz przy kompilacji *wieloprojektowej* (*Make All Projects* i *Build All Projects*). W czasie kompilacji realizowanej „w tle” nie są ponadto aktywne mechanizmy kategorii *Code Insight*, niedostępne są także niektóre opcje menu IDE, jak np. *Project|Options*.

Mimo iż kompilacja „w tle” generalnie przyczynia się do przyspieszenia pracy nad projektem, sama czynność kompilowania kodu źródłowego realizowana jest w tym trybie średnio o 25 procent wolniej w stosunku do „zwykłej” kompilacji pierwszoplanowej. Odbywa się ona bowiem w ramach oddzielnego wątku i wymaga różnorodnych zabiegów synchronizacyjnych przy dostępie do buforów danych, plików modułów itp. Jeżeli spowolnienie to okaże się dla programisty dokuczliwe, może on w ogóle zrezygnować z kompilacji „w tle” (na rzecz „zwykłej” kompilacji pierwszoplanowej), usuwając zaznaczenie opcji *Background compilation* na karcie *Preferences* opcji środowiska (*Tools|Environment Options*).

Pozostałe nowości kompilatora

Nowe modyfikatory funkcji — `__msfastcall` i `__msreturn` — umożliwiają tworzenie bibliotek DLL przeznaczonych do wykorzystania w aplikacjach tworzonych w środowisku Visual C++. Modyfikatory te powodują skompilowanie funkcji zgodnie z konwen-

cją wywołania Microsoftu *fastcall*; użycie przełączników kompilacji -VM i -pm powoduje domyślne kompilowanie każdej funkcji w taki właśnie sposób, bez konieczności jawnego specyfikowania wspomnianych modyfikatorów.

Dyrektywa __declspec została poszerzona o siedem nowych odmian, służących różnorodnym celom — i tak na przykład deklaracja __declspec(naked) powoduje kompilowanie odwołań do danej funkcji (wywołania i powrotu) bez sekwencji wstępnej (prologu) i kończącej (epilogu), natomiast __declspec(noreturn) stanowi informację dla kompilatora, iż brak instrukcji return w ciele danej funkcji nie jest pomyłką, lecz efektem zamierzonym (dotyczyć to może funkcji, których zadaniem jest zakończenie aplikacji); dotychczas funkcje pozbawione instrukcji return powodowały generowanie ostrzeżeń w czasie kompilacji, obecnie ostrzeżenia takie dotyczą jedynie przypadków, gdy bezpośrednio po wywołaniu funkcji o wspomnianej własności znajduje się kod, który mógłby zostać wykonany jedynie po zwróceniu sterowania przez tę funkcję.

Mechanizm rozszerzonego raportowania błędów aktywowany za pomocą opcji *Extended Error Information* na karcie *Compiler* opcji projektu umożliwia uzyskanie dodatkowych informacji związanych z kontekstem sygnalizowanego błędu lub ostrzeżenia, na przykład nazwy funkcji, w czasie analizy której kompilator wykrył sytuację będącą przedmiotem komunikatu. Tę dodatkową informację możemy ujrzeć, klikając symbol „+", poprzedzający zasadniczy komunikat.

Użytecznym w procesie śledzenia może okazać się nowe makro __FUNC__ zastępowane nazwą funkcji, w której się pojawi — w poniższym przykładzie do dziennika śledzenia (*View|Debug Windows| Event Log*) wpisywane są komunikaty „Wejście do TForm1::Button1Click" oraz „Wyjście z TForm1::Button1Click":

```
#define DFUNC_ENTRY OutputDebugString("Wejście do " __FUNC__ )
#define DFUNC_EXIT OutputDebugString("Wyjście z "  __FUNC__ )

void __fastcall TForm1::Button1Click(TObject *Sender)
{
  DFUNC_ENTRY;
  ShowMessage("Wewnątrz funkcji");

  ...

  DFUNC_EXIT;
}
```

Makro __FUNC__ może być również używane w metodach deklarowanych w definicjach klas.

Nowe funkcje konsolidatora

Do opcji projektu dodano nową kartę o nazwie *Advanced Linker*. Zawiera ona kilka nowych opcji, a także opcje istniejące dotychczas jedynie dla wywołania konsolidatora z wiersza poleceń. Jedna z tych opcji — *DLLs to delay load* — udostępnia coś na kształt (znanej z Delphi) eliminacji zbędnego kodu, chociaż w bardziej ograniczonej postaci. Otóż każda z bibliotek, której nazwa wystąpi na liście wspomnianej opcji, włą-

czona zostanie do kodu wynikowego tylko wówczas, gdy konsolidator napotka przynajmniej jedno odwołanie do którejś z funkcji zawartych w tejże bibliotece. Umożliwia to łatwą eliminację bibliotek w danym projekcie niepotrzebnych, lecz charakteryzujących się długim „czasem rozruchu", może być także użyteczne w przypadku tworzenia „okrojonej" wersji aplikacji, w której pewne funkcje nie są w ogóle wywoływane.

Pojawiło się również kilka nowych przełączników konsolidatora, między innymi przełączniki z grupy `-GF`, służące do ustawiania określonych znaczników w module wynikowym, przełącznik `-GD`, powodujący generowanie kompatybilnych z Delphi plików zasobowych **.RC* oraz przełącznik `-ad`, umożliwiający tworzenie 32-bitowych sterowników urządzeń dla Windows.

Umożliwiono także selektywne określenie zestawu ostrzeżeń wykorzystywanych przez konsolidator — służy do tego sekcja *Warnings* na karcie *Linker* opcji projektu.

Podstawowe zasady optymalizacji

Pod pojęciem optymalizacji rozumiemy tu działania zmierzające do ulepszania aplikacji w aspekcie jej najistotniejszych cech z punktu widzenia jej użytkownika — szybkości, wymagań pamięciowych, efektywności operacji dyskowych, obciążenia pasma sieci itp. Ze względu na to, iż matematycznie ścisłe pojmowanie tych cech — zwłaszcza w przełożeniu na kod źródłowy aplikacji — wciąż dalekie jest od kompletności, szeroko pojmowana optymalizacja nosi w sobie znamiona zarówno sztuki, jak i nauki; wymaga bowiem zarówno skomplikowanych dociekań analitycznych, ale również i intuicji projektowej.

Przewidywania Moore'a sprzed trzydziestu lat, iż stopień scalenia procesorów podwajać się będzie co półtora roku, sprawdzają się w całej pełni — procesor Pentium III składa się z ponad 28 milionów tranzystorów; w połączeniu z wzrastającą wciąż szybkością taktowania procesorów oznacza to nieustanny wzrost ich mocy obliczeniowej. Dodając do tego coraz pojemniejsze pamięci RAM, coraz pojemniejsze i szybsze dyski, coraz szybsze karty wideo i płyty CD-ROM, otrzymujemy coraz to potężniejsze komputery — po cóż więc w ogóle kłopotać się optymalizacją aplikacji, skoro (wydawałoby się) wystarczy tylko trochę poczekać na odpowiednio szybki komputer?

Pomijając już kwestię ograniczoności technologii (skończona prędkość rozchodzenia się sygnałów, skończone rozmiary molekuł itp.), należy uświadomić sobie oczywisty fakt, iż zawsze istnieć będą problemy, których złożoność przekraczać będzie możliwości dostępnego sprzętu o kilka rzędów. W latach świetności komputerów *mainframe* przeważająca większość takich problemów wywodziła się z różnych gałęzi fizyki („fizycy potrafią zarżnąć najszybszy nawet superkomputer"), obecnie rolę „poligonów doświadczalnych" w testowaniu możliwości komputerów przejęły coraz efektowniejsze gry komputerowe, jak również profesjonalne narzędzia obróbki obrazu i dźwięku — wymagają one zarówno szybkich procesorów i pojemnych pamięci, jak również realistycznej grafiki i nie zniekształconego dźwięku.

W sytuacji, gdy dana aplikacja zaczyna sprawiać problemy z efektywnością — na przykład szybkość jej działania daleka jest od zadowalającej, zaś z powodu zbyt małej pamięci

RAM komunikacja z plikiem wymiany zdaje się przybierać formę zgoła „konwulsyjną" — dla zaradzenia temu niekorzystnemu stanowi rzeczy możliwe są dwa rodzaje postępowania: rozbudowa sprzętu i optymalizowanie oprogramowania. Rozwiązania sprzętowe charakteryzują się ograniczonym polem manewru, trudne są w optymalizacji, a dodatkową ich barierą są koszty rosnące gwałtownie wraz z ich sprawnością. Nie należy ich mimo to lekceważyć, niekiedy bowiem niezbyt kosztowne zabiegi — jak np. rozbudowa pamięci operacyjnej z 16 MB do 128 MB — okazać się mogą w konsekwencji wręcz zbawienne. Optymalizacja zorientowana sprzętowo jest tematem na tyle złożonym, iż wartym odrębnej obszernej książki, w tym rozdziale zajmiemy się więc wyłącznie działaniami optymalizacyjnymi o charakterze programowym — czyli takimi, które wykonać może programista w związku jedynie z kodem źródłowym aplikacji, bez ingerowania w istniejącą platformę sprzętową czy nawet system operacyjny.

Przystępując do optymalizacji aplikacji — a dokładniej: określonej postaci tej aplikacji — należy najpierw wyznaczyć sobie cele, które chcemy dzięki tej optymalizacji osiągnąć. Podstawową tego przesłanką powinien być daleko posunięty realizm, który rozumieć możemy co najmniej dwojako. Po pierwsze, nie należy stawiać sobie celów *niemożliwych* do zrealizowania: skompresowanie w stosunku 20:1 pełnoekranowej, godzinnej animacji posługującej się 24-bitowym kolorem nie da się żadną miarą zrealizować na współczesnych komputerach w czasie krótszym od jednej minuty, bez względu na to, którego ze znanych algorytmów kompresji by nie użyć. Po drugie — należy być świadomym *ograniczeń*, w związku z którymi w ogóle podejmuje się optymalizację: nie ma na przykład sensu zmniejszanie (za wszelką cenę) rozmiaru pliku wynikowego poniżej 1 megabajta, jeżeli użytkownik zadowoli się możliwością zmieszczenia tegoż pliku na dyskietce 1,44 MB. Nie ma również sensu marnotrawienie czasu i zasobów na windowanie (za wszelką cenę) szybkości aplikacji postrzeganej przez użytkownika jako wystarczająco efektywna.

Zagadnienie sensowności optymalizacji posiada jeszcze jeden istotny wymiar. Otóż większość typowych aplikacji zdaje się wykazywać objawy swoistej „lokalności", zgodnie z którymi mała część (10 – 25 procent) kodu aplikacji realizowana jest przez większość (80 – 99 procent) czasu jej wykonania[3]. Jest oczywiste, iż właśnie takie fragmenty kodu są idealnymi kandydatami dla wszelkich optymalizacji szybkości wykonania, bowiem każde, najdrobniejsze nawet usprawnienie przynosi wówczas zwielokrotnione efekty. Do problemu tego powrócimy w dalszej części rozdziału.

Należy być również świadomym faktu, iż optymalizowanie aplikacji stanowi doskonałą okazję do wprowadzenia w jej kod rozmaitych błędów. Kod bardziej optymalny to niejednokrotnie kod mniej przejrzysty i mniej oczywisty, ponadto — generalnie — nowo wprowadzony kod to przecież kod *nieprzetestowany*.

Pisząc o optymalizacji, ograniczyliśmy się dotychczas głównie do dwóch jej aspektów, mianowicie szybkości i „pamięciożerności". Tymczasem najważniejszym celem optymalizowania aplikacji jest uczynienie jej lepszą z punktu widzenia użytkownika, tak więc postrzegając przysłowiowe drzewa ,nie możemy tracić lasu z pola widzenia; poza

[3] Przejawy tej zasady spotkać można w wielu dziedzinach pozainformatycznych, i tak np. w ekonomii jest ona znana pod nazwą „zasady Pareto" — *przyp. tłum.*

szybkością i zajętością pamięci każdy projekt charakteryzuje się innymi jeszcze, istotnymi dla użytkownika, cechami wśród których wymienić należy między innymi:

- łatwość utrzymywania i konserwacji;
- łatwość testowania;
- użyteczność;
- możliwość wielokrotnego wykorzystania (ang. *reusability*);
- niezawodność;
- skalowalność;
- przenośność;
- łatwość obsługi;
- bezpieczeństwo.

Wymienione cechy mogą posiadać zróżnicowany stopień istotności, zależnie od konkretnego projektu, zawsze jednak należy określić ich priorytety i podporządkować im wszelkie działania optymalizacyjne — jeżeli przykładowo najważniejszymi cechami tworzonej aplikacji mają być szybkość i oszczędność pamięci, należy uwzględnić to jak najwcześniej na etapie jej projektowania. Im głębszy poziom optymalizacji, tym trudniejsze jej przeprowadzanie; w skrajnym wypadku czas spędzony na niskopoziomowym optymalizowaniu kodu może okazać się stracony, na przykład z powodu zmiany koncepcji projektowej i użycia innego algorytmu.

Wreszcie — wszelkie zmiany wprowadzane do kodu w związku z optymalizacją powinny być dokumentowane, zarówno co do funkcji spełnianych przez (zmodyfikowany) kod, jak i celu wynikającego z optymalizacji; wskazane jest także zachowanie pierwotnej wersji kodu w celu ewentualnego porównania jej z wersją zoptymalizowaną w przypadku, gdy pojawią się problemy.

Optymalizacja szybkości wykonania aplikacji

Nader często aplikacje bywają oceniane na podstawie swej szybkości — również pod względem reagowania na polecenia użytkowników — nic więc dziwnego, iż to właśnie szybkość aplikacji jest zazwyczaj podstawowym przedmiotem optymalizacji. Optymalizacja taka staje się konieczna wobec rosnącej wciąż złożoności tworzonych programów i wzrastających rozmiarów przetwarzanych danych z jednej strony, a ograniczonych możliwości procesorów z drugiej. Optymalizacja przekładu tworzonego przez kompilator nie jest co prawda w stanie konkurować z możliwościami projektantów aplikacji w tym względzie, ale — jak zobaczymy w dalszej części rozdziału — i ona może w wydatnym stopniu przyczynić się do szybkości wykonania programu. Kod tworzony przez (optymalizujący) kompilator C++Buildera może pod względem efektywności konkurować z produktami kompilatorów Delphi i Visual C++. Notabene porównywanie „jakości” kompilatorów pod względem efektywności tworzonego kodu jest zadaniem trudnym, a publikowane w tym temacie wyniki nie zawsze traktować można poważnie — przykładowo zapewnienie, iż np. dany kompilator generuje kod *pięciokrotnie* szybszy od kompilatora konkurencyjnego są za-

zwyczaj tyleż nieuczciwe, co mylące, eksponują bowiem zazwyczaj aplikacje pewnej szczególnej kategorii i nie mogą być uogólniane na szeroką gamę *typowych* aplikacji.

Optymalizacje wykonywane przez kompilator C++Buildera 5 mogą skutkować ok. 20 – 55-procentowym przyspieszeniem aplikacji, zależnie od ich zróżnicowanych kategorii. Aplikacje takie wymagają zazwyczaj dodatkowej optymalizacji ze strony projektantów — generalnie rzecz biorąc, do najbardziej prawdopodobnych kandydatów w tym względzie zaliczyć można następujące kategorie aplikacji:

- aplikacje wykonujące dużą liczbę skomplikowanych obliczeń matematycznych — symulacje modeli abstrakcyjnych i zjawisk ze świata rzeczywistego, generatory fraktali, realistyczne symulacje graficzne itp.;
- procesy przetwarzające duże ilości danych — programy kompresji danych, programy sortujące, przeszukiwarki i aplikacje szyfrujące;
- programy wspomagające rozwiązywanie problemów, dokonujące złożonych przeszukiwań i oceny rozwiązań pośrednich — symulatory zdarzeń, algorytmy najlepszego dopasowania, konstruktory optymalnych wzorców; jedną z aplikacji tego rodzaju zajmiemy się już za chwilę.

Projektant tworzący aplikację za pomocą C++Buildera ma do dyspozycji szereg środków jej optymalizacji, w szczególności:

- ustawienia optymalizacyjne kompilatora;
- wybór odpowiedniego algorytmu i założeń projektowych;
- niskopoziomowe zmiany kodu;
- zmiany reprezentacji danych;
- „ulepszanie” wygenerowanego kodu;
- łagodzenie skutków nieefektywności.

Ostatnia z wymienionych możliwości wydaje się cokolwiek zagadkowa i nic w tym dziwnego, bowiem w przeciwieństwie do pozostałych nie wpływa bezpośrednio na efektywność pracującej aplikacji, zmniejszając za to *uciążliwość* rozmaitych przejawów nieefektywności — i tak na przykład długotrwała operacja zyska większe zrozumienie użytkownika, jeżeli ten będzie miał możliwość obserwacji stopnia jej zaawansowania oraz przerwania na żądanie, najlepiej z zachowaniem rezultatów dotychczas wykonanej pracy. Środowisko Win32 stwarza dodatkowe ułatwienia pod tym względem, umożliwiając podział procesów na równolegle wykonywane wątki; czasochłonne czynności mogą być więc realizowane przez osobne wątki biegnące „w tle”, podczas gdy użytkownik zachowuje pełną kontrolę nad całością aplikacji.

Tak się niestety składa, iż polepszenie aplikacji pod względem szybkości pociągać może za sobą jej pogorszenie pod innymi względami, na przykład pod względem rozmiaru lub zajętości pamięci. Przykładowo jednym ze sposobów zwiększenia szybkości programu jest przechowywanie obliczonych wyników pośrednich — zamiast obliczać je ponownie, wystarczy pobrać ich gotową wartość, a to z kolei oznacza zwiększone zapotrzebowanie na pamięć. Istnieją jednak sytuacje, w których zabiegi optymalizacyjne

poprawiają zarówno szybkość aplikacji, jak i wykorzystanie pamięci — najczęściej są one skutkiem sięgnięcia po inny algorytm rozwiązujący dany problem. Przekonamy się o tym naocznie już za chwilę, analizując proces optymalizowania przykładowej aplikacji.

Przykład optymalizacji — konstruktor krzyżówek

Nasza przykładowa aplikacja dokonuje układania krzyżówek składających się ze słów (w języku angielskim) pochodzących z zadanej listy, starając się zmaksymalizować jakość tworzonej krzyżówki zgodnie ze szczegółowo określonymi kryteriami. Należy ona do trzeciej z wymienionych wcześniej kategorii, posługując się zaawansowanymi technikami przeszukiwania i oceny rozwiązań.

Każde wykorzystane w krzyżówce słowo warte jest (z tytułu samego wystąpienia) 10 punktów. Punktowane są również litery znajdujące się na przecięciu słów, i tak litera z zakresu `A÷F` warta jest 2 punkty, z zakresu `G÷L` — 4 punkty, z zakresu `M÷R` — 8 punktów, z zakresu `S÷X` — 16 punktów; litera `Y` warta jest 32 punkty, zaś litera `Z` — 64 punkty. Każde ze słów z podanej listy może być użyte co najwyżej raz i może być ulokowane poziomo albo pionowo w siatce o rozmiarze 15×10 pozycji.

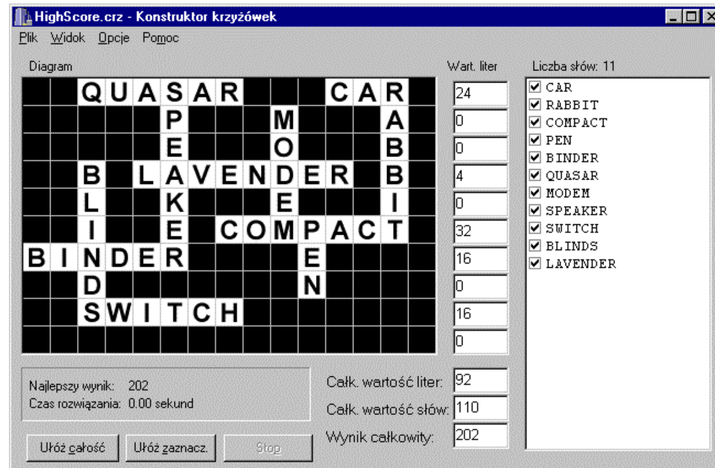
Nie podlegają ocenie słowa znajdujące się wewnątrz innych użytych słów — i tak np. mimo iż słowo `SCARE` zawiera w sobie słowo `CAR`, nie dolicza się z tego tytułu 10 punktów (za nowe słowo) ani też 2+2+8=12 punktów z tytułu wspólnych liter `C`, `A` i `R`. Niezależne wystąpienie słowa `CAR` byłoby jednak warte 10 punktów, zaś jego „przecinanie się" z innymi słowami punktowane byłoby na ogólnych zasadach.

Z założenia lista dostępnych słów zawierać powinna maksymalnie ok. 120 słów o długości od 3 do 15 liter.

Rodowód prezentowanej aplikacji sięga połowy lat osiemdziesiątych, kiedy to jeden z autorów oryginału stworzył ją w BASIC-u na potrzeby konkursu, w którym główna nagroda wynosiła 2000 dolarów. Wobec powolności ówczesnych komputerów, niskiej efektywności samego BASIC-a i nieoptymalnego kodu aplikacja ta nie zdążyła wygenerować na czas nawet częściowego rozwiązania. Obecna jej wersja jest niesamowicie bardziej rozbudowana i zoptymalizowana, chociaż (zdaniem autora) jej efektywność można jeszcze poprawić. Składa się ona z czterech modułów źródłowych i trzech plików nagłówkowych o łącznej objętości ok. 2900 wierszy kodu. Wygenerowane przez nią przykładowe rozwiązanie przedstawia rysunek 4.1.

Na załączonej płycie CD-ROM umieszczone są dwie wersje projektu: w podkatalogu `CrozzleInitial` znajduje się wersja wyjściowa, stanowiąca przedmiot optymalizacji, natomiast podkatalog `CrozzleFinal` zawiera wersję stanowiącą rezultat zastosowania wszystkich zabiegów optymalizacyjnych opisanych w dalszej części rozdziału.

Budowanie krzyżówki może być prowadzone „od zera", poczynając od pustego diagramu, bądź też na bazie istniejącego rozwiązania częściowego (wygenerowanego przez aplikację lub stworzonego „ręcznie"). Startowa postać diagramu wraz z listą dopuszczalnych słów zapamiętana jest w pliku **.CRZ*, który można wczytać za pomocą menu *Plik*. Arenę



Rysunek 4.1. *Przedmiotowa aplikacja z przykładowym rozwiązaniem*

działania aplikacji ograniczyć można do prostokątnego fragmentu diagramu, zakreślając żądany prostokąt kursorem myszy, przy czym ograniczenie to dotyczyć może bądź to całości krzyżówki (żadna litera nie może wówczas wykraczać poza zaznaczony obszar), bądź tylko krzyżowania słów (jakiekolwiek skrzyżowania występować mogą wtedy tylko w zaznaczonym obszarze), zależnie od ustawień w menu *Opcje*. Działanie konstruktora należy wówczas uruchomić za pomocą przycisku *Ułóż zaznaczenie* — naciśnięcie przycisku *Ułóż całość* spowoduje ignorowanie narzuconych ograniczeń.

Każdorazowo, gdy aplikacji uda się znaleźć rozwiązanie lepsze od dotychczasowych (w rozumieniu opisanych przed chwilą kryteriów punktowych), zapisuje ona to rozwiązanie w pliku o nazwie *HighScore.Crz*, który może być użyty jako punkt wyjściowy do kontynuowania obliczeń, bądź też tylko wyświetlony jako świadectwo dotychczasowych „osiągnięć".

Użytkownik może w dowolnej chwili zapisać aktualny stan obliczeń w pliku **.crz*, używając opcji *Zapisz* i *Zapisz jako* menu *Plik* — co może sprawiać niejaką trudność w sytuacji, gdy stan diagramu nieustannie się zmienia; co prawda kliknięcie opcji *Plik* na pasku menu głównego „zamraża" wyświetloną zawartość diagramu (o ile jest ona w ogóle widoczna), jednak to, co widoczne jest wówczas na planszy, niekoniecznie musi odpowiadać stanowi wewnętrznych struktur programu.

Wykładnicza złożoność algorytmu

Optymalizację naszej aplikacji przeprowadzać będziemy stopniowo, etapami, odnotowując na każdym etapie względne przyspieszenie zarówno w stosunku do etapu poprzedniego, jak i do postaci początkowej. Prezentowane tu wyniki uzyskane zostały na komputerze z procesorem Pentium II 266 MHz i 128 MB pamięci RAM, wyposażonym w system operacyjny Windows 2000 Professional; każdy wynik jest wartością średnią uzyskaną z trzykrotnego powtórzenia pomiaru.

Pojedynczy przebieg podlegający pomiarowi polegał na ułożeniu najlepszego rozwiązania na podstawie listy 11 słów zawartych w pliku *RunComplete.crz*, znajdującym się na załą-

czonej płycie CD-ROM; przebieg ten startował z pustego diagramu i dokonywał sprawdzenia wszystkich możliwych kombinacji. Dla zobrazowania sposobu pracy programu załączyliśmy również plik *RunPartial.crz*, zawierający 115 słów; przeanalizowanie *każdego* z możliwych rozwiązań w tak dużym zestawie słów przekracza możliwości współczesnych komputerów PC i możemy zadowolić się co najwyżej którymś z rozwiązań częściowych.

Wydawałoby się, iż nic prostszego, jak sprawdzić *wszystkie* możliwe kombinacje i wybrać z nich tę najlepszą; zważywszy jednak olbrzymią liczbę tych kombinacji, łatwo dojść do wniosku, iż prościej powiedzieć, a znacznie trudniej wykonać. Tabela 4.1 przedstawia czas kompletnej analizy zestawów składających się z 5 – 11 słów, wybranych z pliku *RunComplete.crz*; zawiera ona również liczbę analizowanych w każdym zestawie rozwiązań i wynik punktowy najlepszego rozwiązania.

Tabela 4.1.
Wynik analizy przykładowych zestawów słów

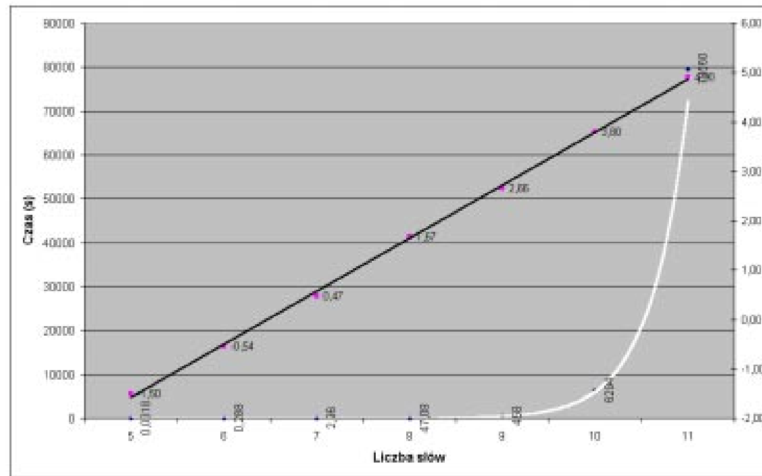
Liczba słów	Czas analizy (s)	Liczba kombinacji	Liczba rozwiązań dopuszczalnych	Najlepszy wynik (pkt.)
5	0,0318	3.577	1.492	90
6	0,288	31.257	9.892	102
7	2,96	317.477	101.604	120
8	47,1	4.765.661	1.192.436	146
9	458	44.057.533	11.123.772	164
10	6.294	554.577.981	152.343.008	190
11	79.560	ponad 6 mld	ponad 1,9 mld	208

Pierwszą rzucającą się w oczy rzeczą jest gwałtowny wzrost czasu obliczeń wraz ze zwiększaniem liczby słów; dokładniejsza analiza pozwala stwierdzić ok. 11-krotny (!) wzrost czasu analizy wskutek dołączenia kolejnego słowa do zestawu. Złożoność czasowa algorytmu ma więc charakter *wykładniczy*, co lepiej zobaczyć można na rysunku 4.2, prezentującym zależność czasu obliczeń od liczby słów w skali liniowej i logarytmicznej.

Ekstrapolując otrzymane wyniki, można obliczyć, iż niezoptymalizowana wersja aplikacji potrzebowałaby ok. 50 lat na uporanie się z 15 słowami, 17 słów zajęłoby jej ponad 7000 lat, zaś wypróbowanie wszystkich rozwiązań ze zbiorem 20 słów trwałoby ponad 10 milionów lat. I nawet wzrastająca wciąż moc obliczeniowa procesorów niewiele by tu pomogła — procesorowi o szybkości *biliona* (10^{12}) MHz sprawdzenie wszystkich kombinacji z użyciem 30-wyrazowej listy i tak zajęłoby ponad 100 milionów lat!

Dane te dotyczą oczywiście naszego konkretnego diagramu 15×10 kratek i ze względu na jego ograniczony rozmiar zależność liczby możliwych kombinacji od liczby słów zacznie w pewnym momencie tracić swój wykładniczy charakter z prostej przyczyny — braku dostatecznego miejsca. Wydaje się, iż możliwość ułożenia poprawnej krzyżówki w tak małym obszarze kończy się gdzieś w okolicy 40 słów. Mimo to wyczerpujące przeanalizowanie układu 115 słów i tak zajęłoby obecnym komputerom znacznie więcej czasu, niż upłynęło dotąd od Wielkiego Wybuchu!

Rysunek 4.2.
Wykładnicza zależność czasu analizy od liczby słów



Pozostańmy więc przy naszej liście ograniczonej do 11 słów, odnotowując w rankingu wartość wyjściową:

pierwotny czas wykonania: 79.560 sekund dla 11 słów.

Opcje projektu wpływające na szybkość generowanego kodu

Jak już wcześniej pisaliśmy, najbardziej oczywistym sposobem sterowania efektywnością generowanego kodu jest odpowiednie ustawienie opcji kompilatora i konsolidatora. Aby zmaksymalizować szybkość aplikacji, należy w związku z tym dokonać następujących ustawień opcji projektu:

- na karcie *Compiler* należy kliknąć przycisk *Release*, co spowoduje automatyczne ustawienie większości opcji wpływających na szybkość aplikacji — strategia optymalizacji (sekcja *Code optimization*) ustawiona zostaje na *Speed*, wyłączone zostają wszystkie opcje w sekcji *Debugging* oraz opcja *Stack frames* w sekcji *Compiling*;
- na karcie *Advanced Compiler* należy wybrać *Pentium Pro* jako procesor docelowy (sekcja *Instruction set*), ustawić wyrównanie danych na granicy wielokrotności 8 bajtów (opcja *Quad word* w sekcji *Data alignment*), ustalić konwencję wywołania (sekcja *Calling convention*) na *Register* i wymusić automatyczną implementację zmiennych rejestrowych (opcja *Automatic* w sekcji *Register variables*). W sekcji *Floating point* należy także wybrać model *Fast* realizacji arytmetyki zmiennoprzecinkowej oraz zrezygnować z ochrony przed wadliwym działaniem instrukcji FDIV we wczesnych modelach Pentium, pozostawiając niezaznaczoną opcję *Correct Pentium FDIV flaw*;
- o ile pozwalają na to uwarunkowania konkretnej aplikacji, należy w sekcji *Exception handling* na karcie *C++* usunąć ew. zaznaczenie opcji *Enable RTTI*, ponadto usunąć zaznaczenie opcji *Enable exceptions* albo zaznaczyć opcję *Fast exception prologs*. Należy ponadto ustawić jako *Smart* model implementacji V-tablic (sekcja *Virtual tables*);

- na karcie *Pascal* należy zaznaczyć opcje *Optimization* i *Aligned record fields* w sekcji *Code generation* oraz usunąć zaznaczenie dwóch pozostałych opcji tej sekcji — *Stack frames* i *Pentium—safe FDIV*. Należy ponadto usunąć zaznaczenie we wszystkich opcjach sekcji *Runtime errors* i *Debugging*;
- na karcie *Linker* należy usunąć zaznaczenie opcji *Create debug information*, *Use dynamic RTL* i *Use debug libraries* (sekcja *Linking*) i wyłączyć generowanie pliku mapowania, wybierając opcję *Off* w sekcji *Map file*. Jeżeli wynikiem konsolidacji ma być biblioteka DLL, należy postarać się wybrać adres bazowy ładowania (*Image base*) tak, by rozpoczynał on dostatecznie duży wolny fragment przestrzeni pamięci wirtualnej aplikacji docelowej (choć nie zawsze jest to możliwe do zrealizowania);
- na karcie *Packages* należy pozostawić niezaznaczoną opcję *Build with runtime packages*;
- na karcie *Tasm* należy wybrać opcję *None* w sekcji *Debug information*;
- na karcie *CodeGuard* należy wyłączyć opcję *CodeGuard Validation*.

Włączenie optymalizacji generowanego kodu powoduje zazwyczaj duże utrudnienia w śledzeniu aplikacji, a to ze względu na przestawianie lub wręcz usuwanie niektórych porcji kodu i danych. Może to skutkować innym działaniem punktów przerwań (*breakpoints*) i innym przebiegiem pracy krokowej w stosunku do tego, czego spodziewa się użytkownik. Stąd wniosek, iż do optymalizowania aplikacji należy przystąpić dopiero po przetestowaniu jej poprawności.

W przypadku naszej aplikacji niektóre z wymienionych opcji nie okazują się mieć żadnego zauważalnego wpływu na jej szybkość, niektóre natomiast — wręcz przeciwnie; w efekcie wskutek li tylko *automatycznej* optymalizacji otrzymujemy całkiem niezły wynik:

obecny czas wykonania: 51.240 sekund;

usprawnienie w tym kroku: 55 proc.;

przyspieszenie globalne: 1,55 raza.

Główny wkład do tego sukcesu (40 punktów procentowych) wnoszą opcje *Speed* z karty *Compiler* i ustawienie *Automatic* w sekcji *Register variables* karty *Advanced compiler*.

Wybór typu procesora na karcie *Advanced Compiler* powinien być dokonany stosownie do architektury komputera, na którym aplikacja ma być wykonywana. Procesory nie są oczywiście kompatybilne „w przód" i kod wygenerowany dla Pentium nie będzie działał na *80386*. W przypadku naszej aplikacji jej szybkość zdaje się być niezależna od wyboru konkretnego procesora.

Mówiliśmy już o tym, iż przeważnie tylko niektóre fragmenty kodu warte są w ogóle optymalizacji, może więc okazać się warte dokonanie opisanych ustawień jedynie w stosunku do wybranych modułów (tzw. *node-level options*). Należy w związku z tym zlokalizować żądany moduł w oknie Menedżera projektu i z menu kontekstowego (uruchamianego prawym kliknięciem ikony symbolizującej ów moduł) wybrać opcję *Edit Local Options*. Ukaże się wówczas okno dialogowe udostępniające podzbiór tych opcji projektu, które mają odniesienie do pojedynczych modułów.

Wykrywanie „wąskich gardeł” aplikacji

Wspominaliśmy już kilkakrotnie o tym, iż typowe aplikacje wykazują tendencję do spędzania znakomitej większości swego czasu wykonania w małych fragmentach kodu. Efektywność takich właśnie fragmentów decyduje w głównej mierze o ogólnej efektywności aplikacji, a więc są one tymi obszarami, w których optymalizacja staje się najbardziej opłacalna. Ich wykrywanie nie jest jednak prostą sprawą — działania prowadzące do tego celu podzielić można z grubsza na trzy następujące kategorie:

- profilowanie kodu;
- elementarny pomiar czasochłonności wybranych fragmentów kodu;
- inspekcja projektu i kodu źródłowego.

Profilowanie

Profilowanie jest czynnością prowadzącą do ustalenia czasochłonności poszczególnych fragmentów kodu, z dokładnością do poszczególnych funkcji lub wręcz poszczególnych wierszy. Większość profilatorów (tak nazywać będziemy programy wykonujące profilowanie kodu źródłowego) ogranicza swe działanie do poziomu funkcji, i w większości przypadków okazuje się to zupełnie wystarczające. Niektóre z profilatorów wymagać mogą ingerencji w kod źródłowy programu, niektóre obywają się bez takich przygotowań. W aplikacjach wielowątkowych możliwe jest niezależne profilowanie poszczególnych wątków.

Spośród wielu dostępnych profilatorów niektóre przystosowane są specjalnie do współpracy z C++Builderem; sztandarowym przykładem takiego profilatora jest Sleuth StopWatch firmy TurboPower Software, stanowiący część pakietu Sleuth QA Suite dostępnego w wersji próbnej (*trial*) pod adresem *http://www.turbopower.com*. Profilator ten wykonuje także funkcję deasemblacji kodu wynikowego, generując jednocześnie informację na temat parowania instrukcji przydatną przy optymalizacji niskopoziomowej. Na potrzeby profilowania naszej aplikacji wykorzystaliśmy jego wersję 1.0, w chwili obecnej dostępna jest już jego wersja 2.0.

Pakiet Sleuth QA Suite 2 zawiera także narzędzie o nazwie Sleuth CodeWatch, podobne w swej istocie do CodeGuard. Służy ono do wykrywania „wycieków” pamięci, ze szczególnym uwzględnieniem wycieków spowodowanych przez VCL; potrafi również wyłapywać niepożądane zapisy do pamięci, a także nieprawidłowe parametry wywołania i wyniki zwracane przez funkcje Win32 API.

Spośród innych dostępnych profilatorów wymienić należy między innymi:

- *QTime* produkcji Automated QA, dostępny w wersjach „Standard” i „Lite”. Współpracuje z C++Builderem w wersjach 3., 4. i 5. Wersja „standard” zawiera wiele interesujących funkcji, między innymi analizę pokrycia kodu (*code coverage*) i śledzenie kodu (*code tracing*). Próbna wersja, wraz z dodatkowymi informacjami, dostępna jest pod adresem *http://www.totalqa.com*.
- *RQ’s Profiler*, niedrogi, lecz wymagający umieszczenia specjalnych makr w kodzie źródłowym i jego konsolidacji ze swymi bibliotekami DLL; udostępnia w tym celu

specjalny edytor. Współpracuje ze wszystkimi wersjami C++Buildera. Dostępny jest w wersji *shareware* pod adresem *http://ourworld.compuserve.com/homepages/rq*.

- *VTune Analyzer* produkcji Intel Corp. udostępnia funkcje profilowania kodu, asystenta o nazwie CodeCoach wspomagającego optymalizację na poziomie kodu źródłowego, profilowanie grafu wywołań funkcji, analizę kodu asemblerowego w celu wykrycia możliwości parowania instrukcji i innych cech charakterystycznych dla poszczególnych procesorów itp. Umożliwia profilowanie zarówno na poziomie funkcji, jak i na poziomie poszczególnych wierszy kodu. Nie jest jednak specjalnie ukierunkowany na współpracę z C++Builderem, jest więc mniej wygodny w użyciu niż profilatory wcześniej wymienione. Dostępny jest w wersji próbnej pod adresem *http://developer.intel.com./vtune/analyzer*.

Przed rozpoczęciem profilowania należy zakończyć funkcjonowanie wszystkich aplikacji, mogących mieć wpływ na szybkość wykonywania aplikacji zasadniczej. Dla uzyskania bardziej wiarygodnych wyników należy profilowanie kilkakrotnie powtórzyć, a otrzymane wyniki uśrednić.

Podstawowym problemem przy profilowaniu kodu — niezależnie od używanego profilatora — są rekursywne wywołania funkcji. Profilatory zorientowane są bowiem raczej na poszczególne *definicje* funkcji (rozumianych jako identyfikowane przez nazwę *fragmenty* kodu źródłowego) niż na ich *wywołania*, skutkiem czego statystyka związana z daną funkcją obejmuje sumarycznie wszystkie jej wywołania, niezależnie od poziomów, na których wystąpiły. Aby więc rozdzielić dane odnoszące się do poszczególnych poziomów, należy użyć pewnego triku, umożliwiającego wyeliminowanie rekursji do zdefiniowanego poziomu włącznie. Trik ten polega na stworzeniu pewnej liczby *kopii* funkcji wywoływanej dotąd rekursywnie i wywoływanie ich w sposób nierekursywny, aż do poziomu zależnego od liczby tych kopii.

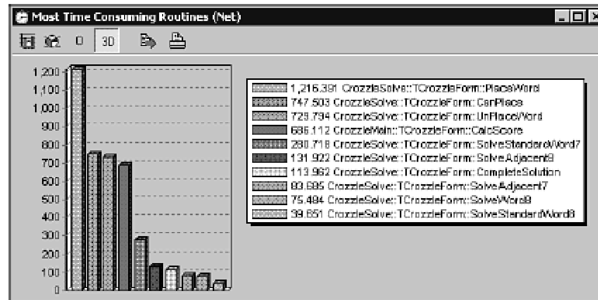
W naszej aplikacji rekursywnemu wywołaniu podlega funkcja `SolveWord()`. W potocznym określeniu jest to rekursja o „odległości 1 i szerokości 3” — funkcja `SolveWord()` wywołuje bowiem trzy funkcje: `SolveFirstWord()`, `SolveAdjacentWord()` i `SolveStandardWord()`, które z kolei wywołują funkcję `SolveWord()`. Rekursywne wywołanie tej ostatniej ma więc charakter pośredni, a owym *pojedynczym* (odległość=1) „poziomem pośredniczącym” jest jedna z funkcji: `SolveFirstWord()`, `SolveAdjacentWord()` lub `SolveStandardWord()`.

W naszej aplikacji zagnieżdżenie rekursywnego wywołania funkcji `SolveWord()` jest większe o jeden od liczby dostępnych słów. Używając więc listy z *siedmioma* słowami i tworząc *osiem* kopii każdej z wymienionych funkcji, wyeliminowaliśmy zupełnie wywołania rekursywne. Tak więc oryginalna funkcja `SolveWord()`wywołuje funkcje: `SolveFirstWord()`, `SolveAdjacentWord()` i `SolveStandardWord()`; każda z tych trzech wywołuje funkcję o nazwie `SolveWord2()`, wywołującą z kolei funkcje: `SolveFirstWord2()`, `SolveAdjacentWord2()` i `SolveStandardWord2()` — i tak dalej. Na wszelki wypadek (gdyby użyto listy o większej liczbie słów) funkcje: `SolveFirstWord8()`, `SolveAdjacentWord8()` i `SolveStandardWord8()`wywołują — tym razem już rekursywnie — funkcję `SolveWord()`; jest to rekursja o odległości 15 (proszę sprawdzić) i szerokości 3. Spreparowanie kodu źródłowego w opisany sposób może wydawać się ogromnie pracochłonne, jednak przy zachowaniu odpowiedniej uwagi okazuje się znacznie prostsze.

Każdą z ośmiu kopii funkcji `SolveWord()` możemy teraz śledzić niezależnie. Należy więc uruchomić Sleuth StopWatch, ustalić profilowanie programu `SolveCrozzle()` w trybie Trigger Mode i zaznaczyć *każdą* z funkcji jako obiekt profilowany. Po skonfigurowaniu StopWatcha należy uruchomić aplikację i wczytać plik **.crz*, zawierający nie więcej niż siedem słów (w przeciwnym razie funkcja `SolveWord()` wywoływana będzie rekurencyjnie i żądana statystyka ulegnie zafałszowaniu).

Graficzną reprezentację „czasochłonności” poszczególnych funkcji, wyświetlaną w oknie StopWatcha przedstawia rysunek 4.3.

Rysunek 4.3.
Przykładowe wyniki profilowania za pomocą Sleuth StopWatcha



Aby uzyskać prawdziwą statystykę dotyczącą każdej z funkcji: `SolveWord()`, `SolveFirstWord()`, `SolveAdjacentWord()` i `SolveStandardWord()` należy dodać do siebie wartości liczby wywołań, czasu netto, średniego czasu wywołania netto i średniego czasu wywołania ogółem każdej z ośmiu kopii odnośnej funkcji. Otrzymane w ten sposób wyniki zawarte są w tabeli 4.2.

Tabela 4.2.
Przykładowe wyniki profilowania na podstawie listy złożonej z siedmiu słów

Funkcja	Liczba wywołań	Czas netto (ms)	Względny czas netto (proc.)	Średni czas wywołania netto (ms)	Czas ogółem (ms)	Średni czas wywołania ogółem (ms)
`PlaceWord`	317462	1216,4	28,78	0,0038	1216,4	0,0038
`CanPlace`	544926	747,5	17,68	0,0014	747,5	0,0014
`UnPlaceWord`	317462	729,8	17,27	0,0023	729,8	0,0023
`CalcScore`	101604	686,1	16,23	0,0068	686,1	0,0068
`SolveStandardWord`	167474	356,6	8,43	0,002	24029,0	0,1435
`SolveAdjacent`	149988	253,95	6,01	0,0017	283,4	0,0019
`SolveWord`	317463	116,7	2,76	0,0004	28663,6	0,0903
`CompleteSolution`	101604	114,0	2,70	0,0011	803,6	0,0079
`SolveFirstWord`	1	0,00	0,00	0,00	4224,6	4224,6

Analizując procentowy udział czasu netto poszczególnych funkcji w ogólnym czasie wykonania aplikacji, stwierdzamy, iż optymalizację powinniśmy rozpocząć od funkcji: `PlaceWord()`, `CanPlace()`, `UnPlaceWord()` i `CalcScore()`.

Elementarny pomiar czasochłonności

Elementarny pomiar czasochłonności poszczególnych fragmentów kodu źródłowego polega na obudowaniu tych fragmentów instrukcjami odczytującymi wskazanie zegara przed rozpoczęciem i po zakończeniu odpowiedniego fragmentu, jak w poniższym przykładzie (wydruk 4.3):

Wydruk 4.3.
Elementarny pomiar czasu wykonania

```
#include <time.h>

void TMyClass::SomeFunction()
{
    clock_t StartTime,
            StopTime;

    // Jakiś kod...

    // Zarejestruj czas rozpoczęcia operacji.
    StartTime = clock();

    // Badany kod...

    // Zarejestruj czas zakończenia operacji.
    StopTime = clock();
    // Ile czasu upłynęło?
    ShowMessage("Zużyty czas: " +
                FloatToStrF((StopTime-StartTime)/CLK_TCK, ffFixed, 7, 2) +
                " sekund.");

    // Ciąg dalszy kodu...
}
```

W przypadku, gdy podlegający pomiarowi fragment kodu wykonuje się zbyt szybko, by można było zmierzyć czas tego wykonania, należy wykonać ów fragment wielokrotnie i otrzymaną różnicę czasową podzielić przez liczbę wykonań — jest to jednak możliwe tylko wówczas, jeżeli fragment ten ma charakter *idempotentny*, co oznacza iż efekt dowolnej liczby jego kolejnych, kompletnych wykonań jest identyczny z efektem pojedynczego kompletnego wykonania.

Wyświetlanie komunikatu za pomocą `ShowMessage()` może być uciążliwe w sytuacji, gdy dana funkcja, zawierająca fragment podlegający pomiarowi, wykonuje się kilkadziesiąt czy kilkaset razy; o wiele wygodniejszy okazuje się wtedy zapis komunikatu do dziennika śledzenia (*Debug Event Log*) wykonywany przez funkcję `OutputDebugString()` lub do odrębnego pliku używanego *ad hoc* w tym celu.

Inspekcja założeń projektowych

Inspekcja ta wykonywana jest na podstawie sformalizowanej dokumentacji projektowej i sprowadza się do wykrywania krytycznych punktów sterowania i przepływu danych. Analizie podlegają fragmenty oryginalnego kodu źródłowego i pseudokodu, diagramy przepływu danych itp. Widoczne stają się wówczas wszelkiego rodzaju intensywnie wykorzystywane fragmenty kodu (np. pętle) i danych (na przykład pliki zawierające dane o kluczowym znaczeniu) i zazwyczaj są one właśnie „wąskimi gardłami" aplikacji.

Inspekcja kodu źródłowego

Inspekcja kodu źródłowego polega na wykrywaniu wszelkich konstrukcji posiadających symptomy złożoności, a więc: pętli, rozgałęzionych skoków, indeksowania i „arytmetyki na wskaźnikach"; należy pamiętać, iż kod zorientowany obiektowo, odwołujący się do dużej liczby „drobnych" metod, również wykorzystuje tę arytmetykę, jednak w sposób niewidoczny bezpośrednio.

Zarówno inspekcja założeń projektowych, jak i inspekcja kodu źródłowego umożliwiają jedynie stawianie hipotez co do czasochłonności „podejrzanych" fragmentów; hipotezy te bywają następnie weryfikowane, na przykład za pomocą opisanego przed chwilą pomiaru elementarnego. Niezależnie jednak od metody wykrywania „wąskich gardeł" aplikacji samo ich wykrycie nie oznacza jeszcze końca pracy, należy bowiem określić *wpływ* każdego z nich na ogólną szybkość wykonywania aplikacji i związaną z tym opłacalność optymalizacji.

Optymalizacja założeń projektowych i algorytmów

W przeciwieństwie do optymalizacji „automatycznej", wykonywanej przez kompilator na podstawie ustawień odpowiednich opcji, dobór właściwych technologii dla realizacji projektu, jak również wybór właściwych algorytmów, daje nieporównywalnie większe pole manewru, o czym przekonamy się już za chwilę.

Właściwe decyzje projektowe

Aby optymalizacja aplikacji mogła rozpocząć się już we wczesnym stadium jej projektowania, konieczne jest dogłębne zrozumienie sposobu realizacji jej podstawowych elementów funkcjonalnych. Nie jest to zadanie łatwe, zwłaszcza w przypadku aplikacji bazujących na obliczeniach rozproszonych lub współpracujących z innymi aplikacjami, niemniej jednak można pokusić się o kilka reguł w tym względzie, dotyczących szczególnie aplikacji dla Windows.

Tak więc wewnątrzprocesowe obiekty — serwery COM są efektywniejsze we współpracy od serwerów zewnętrznych, a zwłaszcza serwerów DCOM zlokalizowanych na innych komputerach. Dokonując skalowania aplikacji bazującej na mechanizmie COM,

warto być może pomyśleć o nowocześniejszych technologiach, jak np. CORBA czy też specjalizowanym interfejsie na podstawie TCP/IP.

W zakresie aplikacji bazodanowych istnieje ponad 20 konkurencyjnych rozwiązań w stosunku do klasycznego BDE; C++Builder 5 implementuje dwa z nich: ADO Express i InterBase Express. Dokonując wyboru konkretnej „maszyny" bazodanowej, należy brać pod uwagę jej możliwości, użyteczność i oczywiście efektywność.

Ogólnie rzecz biorąc, decyzje dotyczące technicznej realizacji funkcjonalnych założeń projektowych są decyzjami trudnymi, wymagającymi wiele namysłu i przede wszystkim rzetelnej informacji. Źródłem tej informacji mogą być liczne grupy dyskusyjne związane z produktami Borlanda, gdzie problemy technologicznego uwarunkowania aplikacji są niezwykle szeroko dyskutowane przez samych zainteresowanych.

Schodząc na nieco niższy poziom szczegółowości, można podać kilka ogólnych zaleceń, którymi powinni kierować się projektanci dążący do optymalności swych aplikacji:

- Unikaj złożoności, nie przesadzaj z modularyzacją aplikacji, rozważ częściową denormalizację baz danych.
- Unikaj zbytniego rozgałęzienia sterowania, w szczególności łańcuchowanych odwołań do rozproszonych obiektów.
- Wwystrzegaj się powolnych technologii, wybierając takie, które najlepiej odpowiadają Twoim aplikacjom.
- Używaj efektywnych narzędzi niezależnych producentów do generowania raportów, kompresji danych, szyfrowania itp.
- Ograniczaj liczbę kontrolek graficznych, zwłaszcza gdy są one często uaktualniane; rozważ ich ukrywanie na czas aktualizacji.
- Powierzaj czasochłonne czynności wątkom drugoplanowym.
- Staraj się zminimalizować obciążenie sieci i intensywność odwołań do dysków; staraj się grupować dane i komunikaty w większe porcje.
- Projektując aplikację dla komputerów wieloprocesorowych, staraj się o równomierne (w miarę możliwości) obciążenie procesorów.
- Ograniczaj wykorzystanie pamięci — w środowisku wieloprogramowym pamięciożerne aplikacje obsługiwane są zazwyczaj z mniejszą efektywnością.

Powyższe zalecenia mają oczywiście charakter orientacyjny, trudno bowiem podać uniwersalny zbiór zasad prawdziwy w odniesieniu do każdej aplikacji.

Nasza przykładowa aplikacja krzyżówkowa jest raczej prostym projektem i wiele z przytoczonych zaleceń po prostu nie ma do niej zastosowania. Jednym z jej aspektów projektowych, które powinny być zoptymalizowane na możliwie wczesnych etapie projektowania, jest kontrolka `TDrawGrid` odpowiedzialna za graficzną reprezentację krzyżówki — domyślnie nie jest ona aktualizowana podczas obliczeń, lecz użytkownik może zmienić ten stan rzeczy za pomocą jednej z opcji menu *View*.

Inna możliwość wczesnej optymalizacji odnosi się do wzajemnej zależności pomiędzy rekurencyjnie wywoływanymi funkcjami, a szczególnie do funkcji `SolveWord()`. W jej początkowym stadium wykonywany jest taki oto fragment kodu:

```
if (WordsPlaced.NumPlaced == 0) {
  SolveFirstWord();
  return(true);
```

Funkcja `SolveWord()` wywoływana jest z poziomu funkcji `SolveCrozzle()`

```
void TCrozzleForm::SolveCrozzle()
{

  ...

  SolveWord();

  ...

}
```

a następne jej wywołania — jak wiadomo z wcześniejszego opisu — mają charakter rekurencyjny, przy czym głębokość rekursji zależna jest od liczby dostępnych słów. Warunek `(WordsPlaced.NumPlaced == 0)` prawdziwy jest jednak tylko przy pierwszym, nierekurencyjnym wywołaniu, gdy diagram jest całkowicie pusty, mimo to jest on (warunek) sprawdzany *niepotrzebnie* na każdym poziomie rekursji. Sprawa wygląda na niebagatelną, wszak dla zestawu ośmiu słów funkcja `SolveWord()` wywoływana jest ponad 6 mld razy (patrz tabela 4.2), a więc jej treść warta jest każdej optymalizacji.

Nieskomplikowana modyfikacja kodu zmienia ten niekorzystny stan rzeczy — należy mianowicie rozdzielić wywołania funkcji `SolveWord()` i `SolveFirstWord()`:

```
void TCrozzleForm::SolveCrozzle()
{

  ...

 if (SolveFromBlank)
  {
    SolveFirstWord();
  }
  else
  {
    ...
    SolveWord();
  }

}
```

i oczywiście usunąć z funkcji `SolveWord()` pierwszy z cytowanych fragmentów. Otrzymujemy dzięki temu zauważalne ulepszenie:

obecny czas wykonania: 49.525 sekund;

usprawnienie w tym kroku: 3,5 proc.;

przyspieszenie globalne: 1,61 raza.

Nie jest to może wynik imponujący, w każdym razie wart nieskomplikowanego bądź co bądź zabiegu.

Wybór odpowiedniego algorytmu

Mianem *algorytmu* określamy sformalizowaną metodę rozwiązywania problemu, posiadającą następujące własności:

- ograniczoność — rozwiązanie uzyskuje się w ograniczonym przedziale czasowym;
- adekwatność — wyniki generowane przez algorytm faktycznie stanowią rozwiązanie danego problemu;
- przewidywalność — dla takich samych danych wejściowych wykonywane są takie same czynności;
- skończoność — przepis rozwiązania problemu zawiera się w skończonej liczbie kroków;
- jednoznaczność — każdy krok algorytmu posiada ściśle zdefiniowane znaczenie.

Dla niemal każdego problemu rozwiązywanego drogą zautomatyzowanych obliczeń istnieje kilka algorytmów różniących się czasem wykonania, wymaganiami pamięciowymi itp. Typowym tego przykładem jest czynność sortowania, dla której istnieją zarówno proste pojęciowo, lecz mało efektywne algorytmy sortowania bąbelkowego oraz sortowanie przez wstawianie i wybieranie, jak również efektywne, lecz bardziej skomplikowane algorytmy sortowania szybkiego, sortowania przez łączenie i sortowania stogowego. Różnicę w funkcjonowaniu trzech wybranych algorytmów sortowania można zaobserwować, uruchamiając projekt *Threads.bpr*, znajdujący się w podkatalogu *Examples\Apps\Threads* lokalnej instalacji C++Buildera 5.

Efektywność poszczególnych algorytmów sortowania zależy zresztą (w różnym stopniu) od danych wejściowych — ich zestawu i uporządkowania. Z posortowaniem 115 losowo uporządkowanych elementów algorytm sortowania szybkiego (*Quicksort*) radzi sobie najszybciej, zaś algorytm sortowania bąbelkowego (*Bubblesort*) — najwolniej; jeżeli jednak elementy wejściowe są już ustawione w żądanej kolejności, sortowanie bąbelkowe jest dla nich wyraźnie szybsze od sortowania szybkiego. Dla małej liczby elementów (mniejszej niż 10) sortowanie bąbelkowe jest szybsze od sortowania szybkiego niezależnie od stopnia uporządkowania tych elementów[4].

Zjawisko wpływu danych wejściowych na efektywność zastosowanego algorytmu daje się w mniejszym lub większym stopniu uogólnić na większość algorytmów; w aplikacji przetwarzającej zróżnicowane zestawy danych, kiedy szybkość jest czynnikiem krytycznym, należy więc zastanowić się nad możliwością implementacji *kilku* algorytmów rozwiązujących dany problem.

---

[4] Algorytm sortowania *stogowego* (*Heapsort*) wykazuje za to znikomą wrażliwość na stopień uporządkowania danych wejściowych — *przyp. tłum.*

Zależność efektywności danego algorytmu od rozmiaru danych wejściowych (dokładniej — asymptotyczne zachowanie się tej efektywności przy rozmiarze danych zmierzającym do nieskończoności) wygodnie jest wyrażać przy użyciu tzw. notacji „dużego O”[5] — i tak czas rozwiązywania problemu przez algorytm o złożoności *O(N)* jest (asymptotycznie) proporcjonalny do rozmiaru danych wejściowych, algorytmy o złożoności *O(N2)* rozwiązują problem w czasie proporcjonalnym do kwadratu jego rozmiaru (złożoność taką posiadają „proste” algorytmy sortowania); nowoczesne metody sortowania — sortowanie szybkie i stogowe — są algorytmami o (średniej) złożoności *O(N*logN)*.

W przypadku *przybliżonego* rozwiązywania problemu różne algorytmy mogą dawać zróżnicowaną dokładność. Przykładem takich przybliżonych konstrukcji jest rysowanie „okrągłych” figur geometrycznych na „skwantowanym” do pikseli ekranie komputera — dla figur o dużych rozmiarach szybkie interpolacyjne algorytmy produkują z reguły kształty mniej wierne niż powolne algorytmy, opierające się na oryginalnym równaniu analitycznym danej figury.

Problematyce algorytmiki i w ogóle rozwiązywania problemów za pomocą komputera poświęcono w ostatnich dziesięcioleciach olbrzymią ilość publikacji książkowych, artykułów w czasopismach oraz dokumentów elektronicznych. Autorzy oryginalnego wydania niniejszej książki polecają szczególnie następujące pozycje:

- *Numerical Recipes in C* — publikacja dostępna w formie pliku *.PDF* pod adresem *http://www.ulib.org/webRoot/Books/Numerical_Recipes*;
- *Stony Brook Algorithm Repository* — zestaw algorytmów dla wielu problemów, wraz z kodem źródłowym, dostępny pod adresem *http://www.cs.sunysb.edu/~algorith*;
- *Object-Oriented Numerics* (*http://oonumerics.org*) to biblioteka klas C++ o nazwie Blitz++, obejmująca takie zagadnienia, jak: wektory i macierze rzadkie, generatory liczb pseudolosowych, małe wektory i tablice itp. Kod źródłowy dostępny jest na zasadzie *Open Source*;
- trzy źródła algorytmów związanych z tworzeniem gier i problemami graficznymi: *http://www.gamedev.net*, *http://www.magic—software.com* i grupa dyskusyjna *comp.graphics.algorithms*;
- *Dr. Dobb's Journal* (*http://www.ddj.com*) to zbiór nieregularnie ukazujących się artykułów na temat algorytmiki. Możliwy jest zakup pojedynczych artykułów bądź płyty CD-ROM, zawierającej publikacje z ostatnich 11 lat (za ok. 80 dolarów);
- Cormen, Leiserson, Rivest *Introduction to Algorithms*, ISBN 0262031418, 1990;
- Sedgewick *Algorithms in C++*, ISBN 0201510596, 1992;
- D. Knuth *The Art of Computer Programming: Sorting and Searching*, ISBN 0201896850, 1998.

[5] Dokładną definicję symbolu O(N) i innych podobnych symboli, jak również wiele interesujących zagadnień związanych z algorytmami, znajdzie czytelnik w książce *Algorytmy i struktury danych z przykładami w Delphi*” (Helion, Gliwice 2000) — *przyp. tłum.*

Z wydawnictw w języku polskim poświęconych algorytmice godnymi polecenia wydają się następujące pozycje:

- R. Stephens *Algorytmy i struktury danych z przykładami w Delphi*, wyd. pol. Helion 2000;
- D. Harel *Rzecz o istocie informatyki — algorytmika*, wyd. pol. WNT, 1992;
- Z. Michalewicz *Algorytmy genetyczne + struktury danych = programy ewolucyjne*, wyd. WNT 1996;
- M.M. Sysło, N. Deo, J.S. Kowalik *Algorytmy optymalizacji dyskretnej z programami w języku Pascal*, wyd. PWN 1995;
- E.M. Reingold, J. Nievergelt, N. Deo *Algorytmy kombinatoryczne*, wyd. pol. PWN 1985;
- T. Kręglewski, T. Rogowski, A. Ruszczyński, J. Szymanowski *Metody optymalizacji w języku FORTRAN*, wyd. PWN 1984;
- A. Bartkowiak *Podstawowe algorytmy statystyki matematycznej*, wyd. PWN 1979;
- J. Kucharczyk, M. Sysło *Algorytmy optymalizacji w języku Algol 60*, wyd. PWN 1977;
- J. Kucharczyk *Algorytmy analizy skupień w języku Algol 60*, wyd. PWN 1982;
- S. Paszkowski *Zastosowania numeryczne wielomianów i szeregów Czebyszewa*, wyd. PWN 1975.

Usprawnianie algorytmów

Podobnie jak dany problem może być rozwiązany przez kilka różnych algorytmów, tak i konkretny algorytm może być w różny sposób zakodowany w konkretnym języku programowania. W przypadku samodzielnej implementacji algorytmu istnieje zazwyczaj duża swoboda w realizacji wielu szczegółów, także standardowa biblioteka szablonów (STL) zawiera wiele szablonów zoptymalizowanych pod kątem określonych typów danych. Szczegółowe informacje na ten temat znajdują się w systemie pomocy C++Buildera.

Obszerne informacje na temat optymalizacji kodu w C++, również w zakresie biblioteki STL, znajdują się na stronie *www.tantalon.com/pete/cppopt/main.htm*.

Jedną z podstawowych technik przyspieszania algorytmów jest unikanie powtórnego obliczania tych samych wartości — raz obliczone powinny być przechowywane w statycznych tablicach; pobranie gotowej wartości z tablicy jest na ogół znacznie szybsze od obliczania tejże wartości, podobnie jak w przypadku ręcznych obliczeń odczytanie gotowej wartości z tablicy jest szybsze od jej obliczania ołówkiem na papierze.

Okazją do zastosowania tej techniki jest obliczanie kodu kontroli cyklicznej CRC, chroniącego integralność danych. Obliczanie kodu CRC dla konkretnego obszaru danych sprowadza się do obliczania wartości pewnego wielomianu dla każdego bajta tegoż obszaru, dla dużych obszarów będzie więc korzystne wstępne obliczenie wszystkich war-

tości tego wielomianu dla każdej z 256 możliwych wartości bajta, przechowanie tych wartości w 256-pozycyjnej tablicy i pobieranie ich w miarę potrzeby. Nie ma jednak nic za darmo i również w tym wypadku za usprawnienie płacimy cenę w postaci wstępnego obliczenia wszystkich możliwych wartości i w ogóle miejsca potrzebnego na tę tablicę. Nie jest to bynajmniej wyjątkiem — bardzo często projektant staje wobec słynnego dylematu „czas — pamięć", mając możliwość poprawienia szybkości programu za cenę zajęcia dodatkowej pamięci lub też zmniejszenia jego pamięciożerności za cenę wydłużenia czasu obliczeń[6].

Podobną zasadę zastosować można do kodu charakteryzującego się rozbudowanymi rozgałęzieniami: zamiast wielokrotnego sprawdzania warunków wykorzystać można tablicę, zawierającą adresy funkcji odpowiednich dla każdego warunku. Poniższy fragment rysujący wielokąt o kształcie stosownym do podanego kodu:

```
switch(NumSides) {
case 1: Circle(x, y, r); break;
case 2: PolyError(x, y, r); break;
case 3: Triangle(x, y, r); break;
case 4: Square(x, y, r); break;
case 5: Pentagon(x, y, r); break;
}
```

zapisać można z użyciem tablicy przeglądowej:

```
void (*PolyFunctions[5])(int, int, int) = {
      Circle, PolyError, Triangle, Square, Pentagon
}

...

PolyFunctions[NumSides](x, y, r);
```

Ogólnie — jeżeli wybór konkretnej funkcji uzależniony jest od wartości N zmiennych boolowskich, to wybór ten można zrealizować na podstawie 2^N-pozycyjnej tablicy przeglądowej. W przypadku trzech zmiennych A, B i C może to wyglądać następująco:

```
void (*MyFunctions[8])()=
{
   MyFunction0,  // A=false, B=false, C=false
   MyFunction1,  // A=false, B=false, C=true
   MyFunction2,  // A=false, B=true,  C=false
   MyFunction3,  // A=false, B=true,  C=true
   MyFunction4,  // A=true,  B=false, C=false
   MyFunction5,  // A=true,  B=false, C=true
   MyFunction6,  // A=true,  B=true,  C=false
   MyFunction7   // A=true,  B=true,  C=true
}
```

W naszej przykładowej aplikacji wykorzystaliśmy tę ideę wielokrotnie. Aby wybrać kolejne słowo z listy jako kandydata do umieszczenia na diagramie, program analizuje każdą literę każdego słowa na liście i wyszukuje wszystkie wystąpienia tej litery w dia-

[6] Konkretne przykłady takich kompromisów opisane są we wspomnianej już książce *Algorytmy i struktury danych z przykładami w Delphi — przyp. tłum.*

gramie. Przy każdym wystąpieniu poszukiwanej litery przeprowadzana jest seria testów, mających na celu sprawdzenie, czy w miejscu tego wystąpienia badane słowo z listy może zostać skrzyżowane ze słowem istniejącym na diagramie. Złożoność tego procesu stanowi wspaniałe pole do optymalizacji.

Po pierwsze, dla każdej litery alfabetu tworzona jest lista jej wystąpień (pozycji) w diagramie; każdorazowo, gdy nowe słowo umieszczane jest na diagramie, listy związane z jego literami są uaktualniane. Dzięki temu w celu znalezienia wszystkich wystąpień żądanej litery wystarczy przetworzyć związaną z nią listę, zamiast przeszukiwać 150 pozycji diagramu.

Po drugie, z każdą „kratką" diagramu związana jest informacja o tym, czy słowo, do którego owa kratka przynależy, ma orientację poziomą czy pionową. Informacja ta ma postać dwóch zmiennych boolowskich `HorizWord` i `VertWord`; dla kratki nie należącej do danego słowa obie te zmienne mają wartość `false`, dla kratki leżącej na przecięciu słów obydwie równe są `true`, dla pozostałych kratek dokładnie jedna z nich równa jest `true`. Dzięki temu, testując daną pozycję diagramu jako kandydata na skrzyżowanie słów, łatwo możemy określić kierunek, w którym ulokować należy ewentualne nowe słowo — jest to ten kierunek, dla którego odpowiednia zmienna (`HorizWord` lub `VertWord`) ma wartość `false`.

Po trzecie, z każdą kratką leżącą na przecięciu słów związana jest informacja o punktacji tego przecięcia (*score*); dla kratek nie leżących na przecięciach wartość ta równa jest zeru. Informacja ta tworzona jest w momencie krzyżowania słów, a dzięki niej obliczenie sumarycznej punktacji związanej z przecięciami wymaga tylko zsumowania punktacji wszystkich kratek, zamiast czasochłonnego wykrywania przecięć i osobnej „wyceny" każdego z nich. Obliczająca punktację funkcja `CalcScore()` jest przecież jednym z „wąskich gardeł" aplikacji, co wynika z tabeli 4.2.

Wykorzystywanie różnego rodzaju tablic przeglądowych jest szczególnym przypadkiem bardziej ogólnego aspektu optymalizacji — unikania obliczeń niepotrzebnych, na przykład dublujących się. Jaskrawym przykładem takiego „dublowania" jest w naszej aplikacji dwukrotne sprawdzanie możliwości skrzyżowania każdej pary słów. Jeżeli na przykład w liście występują słowa JAR i ROD, to najpierw do poziomo (na przykład) ulokowanego słowa JAR dopasowuje się pionowe skrzyżowanie ROD, a nieco później do pionowo ulokowanego słowa ROD tworzy się poziome skrzyżowanie JAR. W obydwu przypadkach wynik jest ten sam, a więc druga próba okazuje się wyraźnie niepotrzebna.

Rozwiązanie tego problemu ma charakter dość skomplikowany, jego istotą jest jednak wprowadzenie pewnego *uporządkowania* słów, jeśli chodzi o kolejność *pobierania* ich do testów; można sformułować w tym względzie dwie dosyć ogólne reguły.

- Dla pustego diagramu: po umieszczeniu na diagramie pierwszego słowa kandydatami na utworzenie z nim skrzyżowania są wyłącznie słowa występujące w liście na dalszych niż ono pozycjach.
- Dla diagramu częściowo zapełnionego: wśród słów występujących na liście po badanym słowie odnajduje się słowo najdawniej umieszczone na diagramie — słowo to wraz z następnymi słowami na liście tworzy zbiór kandydatów do badania na możliwość skrzyżowania z badanym słowem.

Po zaimplementowaniu opisanych usprawnień ponownie zmierzyliśmy czas kompletnej analizy 11-elementowego zbioru słów. Oto wynik:

obecny czas wykonania: 24,33 sekundy;

usprawnienie w tym kroku: 203600 proc.;

przyspieszenie globalne: 3270 razy.

To już nie są żarty — takie usprawnienie oznacza skrócenie obliczeń np. z 10 lat do jednego dnia! Oto przykład, jak zbędne operacje potrafią sparaliżować każde obliczenie.

W naszej aplikacji kryje się jeszcze jedna, tym razem niezbyt spektakularna możliwość ulepszenia — w celu określenia, czy dana kratka diagramu leży na skrzyżowaniu słów, bada się związane z nią dwie zmienne `HorizWord` i `VertWord`, gdy tymczasem wystarczyłoby badanie, czy *punktacja* związana z daną kratką jest niezerowa. Wprowadzając tę modyfikację, otrzymujemy niewielkie, choć zauważalne przyspieszenie:

obecny czas wykonania: 23,68 sekundy;

usprawnienie w tym kroku: 2,7 proc.;

przyspieszenie globalne: 3360 razy.

Jest to znacznie więcej, niż zdziałać może nawet najbardziej inteligentna optymalizacja automatyczna — wobec czego materialnych kształtów nabiera stwierdzenie ze wstępu do rozdziału o przewadze optymalizującego programisty nad optymalizującym kompilatorem.

Wysokopoziomowe techniki optymalizowania generowanego kodu

Istnieje wiele technik optymalizowania kodu źródłowego aplikacji pod kątem efektywności kodu generowanego przez kompilator. Należy przy okazji pamiętać o tym, iż kod „wyglądający" optymalnie z punktu widzenia C++ niekoniecznie musi skutkować efektywnym kodem wynikowym, a także o tym, iż efektywność kodu źródłowego niekoniecznie idzie w parze z jego czytelnością.

Specyfika nowoczesnych procesorów

Stopień złożoności współczesnych procesorów znajduje się w ścisłej relacji ze złożonością kodu, jaki przychodzi im wykonywać. Co prawda jedyną *bezpośrednią* możliwością kontroli kodu wynikowego generowanego z programu napisanego w języku wysokiego poziomu są wszelkiego rodzaju „wstawki" asemblerowe i nie inaczej jest w C++, jednakże niektóre postaci kodu źródłowego dają kod bardziej efektywny z punktu widzenia architektury konkretnego procesora. W następnych punktach zakładamy, iż procesorem,

pod kątem którego optymalizujemy kod naszej aplikacji, jest Pentium II — znakomita większość rozpatrywanych tu jego cech odnosi się także do równorzędnych procesorów AMD serii K.

Przewidywanie rozgałęzień

Najważniejszą cechą sprzętowej optymalizacji procesora Pentium II jest możliwość statycznego i dynamicznego *przewidywania rozgałęzień* (ang. *branch prediction*). W kodzie procesorów serii x86 wyróżnić możemy trzy rodzaje skoków: bezwarunkowe, warunkowe w przód i warunkowe w tył. W odniesieniu do kodu w C++ instrukcje skoków bezwarunkowych generowane są bezpośrednio w wyniku przekładu instrukcji: `continue`, `break`, `goto` i pośrednio w wyniku instrukcji: `if`, `?:` i `switch`. Skoki warunkowe w przód generowane są przez instrukcje: `if`, `?:`, `switch` i `while`, zaś skoki warunkowe w tył — przez instrukcje: `for`, `while` i `do`.

Przyjrzyjmy się poniższemu — źle zaprojektowanemu — fragmentowi programu, dokonującemu rysowania okręgów o różnych promieniach:

Wydruk 4.4.
Przykład złego zaprojektowania rozgałęzień

```
if (Radius > MaxRadius) {
    return(false);
}

if (Radius < 50 && Filled) {
    CircleSmallAlgorithmFilled(X, Y, Radius);
}

if (Radius < 50 && !Filled) {
    CircleSmallAlgorithmOpen(X, Y, Radius);
}

if (Radius >= 50 && Filled) {
    CircleLargeAlgorithmFilled(X, Y, Radius);
}

if (Radius >= 50 && !Filled) {
    CircleLargeAlgorithmOpen(X, Y, Radius);
}

return(true);
```

Styl kodu wynikowego generowanego w wyniku przekładu powyższego fragmentu można schematycznie przedstawić w następującej postaci:

Wydruk 4.5.
Schemat przekładu fragmentu z wydruku 4.4

```
if (Radius <= MaxRadius) {
    goto ifb;
}
return(false);

ifb:
if (Radius >= 50) {
    goto ifc;
}
```

```
if (!Filled) {
    goto ifc;
}

CircleSmallAlgorithmFilled(X, Y, Radius);

ifc:

// itd... pozostałe trzy instrukcje if są podobne.
```

Zwróć uwagę, iż warunki w instrukcjach skoku generowanych przez kompilator są odwróceniem warunków w instrukcjach `if` oryginalnego kodu — warunkowe *wykonanie* fragmentu kodu w C++ jest bowiem realizowane przez kompilator jako warunkowe *przeskoczenie* przekładu tego fragmentu. Gdy występują instrukcje `else`, dla każdego bloku `else` lub `if` (z wyjątkiem ostatniego) generowany jest skok bezwarunkowy przeskakujący „resztę" przekładu instrukcji `if`.

Procesor, napotykając wykonywaną już wcześniej instrukcję skoku, stara się przewidzieć rezultat jej wykonania (skok lub brak skoku) na podstawie rezultatów jej wcześniejszych wykonań — 512 ostatnio wykonywanych instrukcji skoku przechowywanych jest w specjalnej pamięci asocjacyjnej nazywanej *Branch Target Buffer* (BTB)[7]; nazywamy to przewidywaniem dynamicznym, bowiem informacja będąca podstawą wnioskowania, przechowywana na dwóch bitach (dla każdej pozycji BTB), jest nieustannie aktualizowana. Gdy procesor napotka instrukcję skoku nie posiadającą swego odpowiednika w tablicy BTB, stosuje przewidywanie *statyczne* — w stosunku do skoków warunkowych i warunkowych w tył przewiduje się ich wykonanie, dla skoków warunkowych w przód — brak skoku. Nietrafione przewidywanie oznacza stratę kilku cykli zegarowych, procesor musi bowiem oczyścić kolejkę instrukcji pobranych z wyprzedzeniem w nadziei ich wykonania.

Aby przystosować nieco nasz kod z wydruku 4.4 do mechanizmu przewidywania rozgałęzień, musimy wykonać w nim dwie podstawowe zmiany. Po pierwsze, zredukujemy w nim liczbę skoków — oznacza to generalnie mniejszą liczbę przewidywań, również tych nietrafionych, a także mniejszy rozmiar historii skoków do zapamiętania. Druga zmiana polega na przekształceniu *niezależnych* instrukcji `if` w instrukcje *zagnieżdżone* oraz dodaniu frazy `else`. W dotychczasowej wersji kodu, jeżeli promień okręgu (`Radius`) przekracza wartość `MaxRadius`, wykonywany jest pojedynczy skok warunkowy w przód; w przeciwnym razie może być wykonanych nawet *siedem* skoków warunkowych w przód, nim trafi się na spełniony warunek. Wydruk 4.6 pokazuje, w jaki sposób można tę liczbę zredukować.

Wydruk 4.6.
Zmodyfikowany kod źródłowy generujący mniejszą liczbę skoków

```
if (Radius > MaxRadius) {
    return(false);
} else if (Radius < 50) {
    if (Filled) {
```

[7] Mechanizm przewidywania rozgałęzień opisany jest szczegółowo w książce *Anatomia PC*, wyd. V (Helion, Gliwice 1999) — *przyp. tłum.*

```
        CircleSmallAlgorithmFilled(X, Y, Radius);
    } else {
        CircleSmallAlgorithmOpen(X, Y, Radius);
    }
} else {
    if (Filled) {
        CircleLargeAlgorithmFilled(X, Y, Radius);
    } else {
        CircleLargeAlgorithmOpen(X, Y, Radius);
    }
}

return(true);
```

W sytuacji, gdy `Radius > MaxRadius` nic się tu nie zmienia — wykonywany jest skok warunkowy w przód, jednak zamiast *siedmiu* mamy teraz *trzy* skoki warunkowe w przód i jeden skok bezwarunkowy; ostatnia z kombinacji — `Radius >= 50` i `!Filled` — nie wymaga skoku bezwarunkowego.

Niestety, nie da się zapewnić tak daleko posuniętej kontroli w przypadku instrukcji `switch`.

Kolejna możliwość ulepszenia kodu wiąże się z poprawą przewidywalności skoków. Kiedy kod wykonywany jest po raz pierwszy, stosowane jest przewidywanie statyczne; w stosunku do pierwszej z instrukcji, będącej instrukcją skoku warunkowego w przód, procesor założy jak wiadomo brak skoku, a dokładniej — brak przeskoku instrukcji `return(false)`, czyli jej wykonanie. Jeżeli promień okręgu nie przekracza wartości `MaxRadius`, jest to przewidywanie nietrafione. Jeżeli 95 procent okręgów będzie miało promień mniejszy od `MaxRadius`, kod tej instrukcji nie zagrzeje długo miejsca w tablicy BTB ze względu na dużą liczbę innych instrukcji skoku; gdy w efekcie przyjdzie do ponownego wykonania tej instrukcji, znowu będziemy mieli do czynienia z przewidywaniem statycznym, oczywiście także nietrafionym.

Podobne konsekwencje związane są także z trzecią z instrukcji wydruku 4.6, jeżeli większość okręgów ma promień o długości 50 lub większej.

Przegrupujmy więc instrukcje naszego przykładu tak, by najbardziej prawdopodobny do wykonania fragment znalazł się w obrębie pierwszego bloku `if`, drugi co do prawdopodobieństwa — w pierwszym bloku `else` itd. Kod prezentowany na wydruku 4.7 jest optymalny w sytuacji, gdy promień większości okręgów nie jest mniejszy niż 50 i jednocześnie nie przekracza wartości `MaxRadius`.

Wydruk 4.7.
Reorganizacja instrukcji warunkowych pod kątem optymalności w określonej sytuacji

```
if (Radius <= MaxRadius) {
    if (Radius >= 50) {
        if (Filled) {
            CircleSmallAlgorithmFilled(X, Y, Radius);
        } else {
            CircleSmallAlgorithmOpen(X, Y, Radius);
        }
    } else {
        if (Filled) {
```

```
            CircleLargeAlgorithmFilled(X, Y, Radius);
        } else {
            CircleLargeAlgorithmOpen(X, Y, Radius);
        }
    }
} else {
    return(false);
}

return(true);
```

Ostatnia z możliwości optymalizacji przedstawionego przykładu pod kątem efektywności skoków wiąże się z wyrażeniami logicznymi (boolowskimi) — należy mianowicie zastąpić *boolowskie* operatory `||` i `&&` *bitowymi* operatorami `|` i `&`, tak więc np. wyrażenie `((A && B) || C)` przekształcić należy do postaci `((A & B)|C)`. W przeciwieństwie do wyrażeń boolowskich, realizowanych z użyciem instrukcji skoków, wyrażenia złożone z operacji bitowych realizowane są jako sekwencja obliczeń, co najwyżej zakończona *pojedynczą* instrukcją skoku bezwarunkowego.

Pamięć podręczna (cache)

Procesor Pentium II posiada po 16 KB pamięci podręcznej 1. poziomu (L1) dla kodu i danych. Pobieranie kodu i danych z pamięci podręcznej przebiega znacznie szybciej niż pobieranie ich z pamięci RAM. Programując iteracyjnie powtarzane fragmenty kodu (pętle), należy w miarę możliwości uczynić je tak małymi, by mogły pomieścić się w pamięci podręcznej. Pierwszy „obrót" pętli wykona się wówczas z „normalną" szybkością, gdyż odpowiedni kod musi zostać załadowany do pamięci podręcznej, natomiast kolejne obroty będą już przebiegać znacznie szybciej.

„Strojenie" kodu

Niektóre możliwości optymalizacji kodu w C++ są na tyle uniwersalne, iż ich skuteczność nie jest uzależniona od konkretnych cech architektury procesorów — mają one raczej związek z samym językiem i sposobem traktowania przez kompilator określonych jego konstrukcji. Oto kilka przykładów z tej kategorii:

- unikanie słowa kluczowego `register` — decyzję o tym, które zmienne mają być przechowywane w rejestrach, należy pozostawić kompilatorowi, wybierając opcję *Automatic* w sekcji *Register variables* na karcie *Advanced Compiler* opcji projektu;
- używanie słowa kluczowego `const` — opatrywanie klauzulą `const` tych deklaracji zmiennych oraz parametrów funkcji i metod, których wartość z założenia pozostać ma niezmieniona, umożliwia kompilatorowi dokonywanie dodatkowych optymalizacji;
- przestrzeganie zgodności typów — konwersje pomiędzy różnymi typami danych (np. `double` i `long`) spowalniają proces obliczeniowy i uniemożliwiają kompilatorowi wykonywanie niektórych optymalizacji;

- usuwanie zbędnego kodu — w złożonych aplikacjach rozmaite unowocześnienia mogą czynić niektóre fragmenty kodu niepotrzebnymi, na przykład wskutek zastąpienia przez nowsze funkcje. Takie zbędne fragmenty kodu bezproduktywnie powiększają rozmiar kodu wynikowego, poza tym stanowić mogą istotne utrudnienie dla automatycznej optymalizacji dokonywanej przez kompilator.

Przyjrzyjmy się teraz wybranym zabiegom optymalizacyjnym tej kategorii.

Używanie funkcji wstawianych (inline)

Wywoływanie „zwykłych" funkcji wiąże się z pewnymi dodatkowymi czynnościami, polegającymi na: przygotowaniu parametrów (poprzez umieszczenie ich na stosie i późniejsze zdjęcie bądź przez przekazanie ich w rejestrach), wykonaniu instrukcji wywołania (CALL), realizacji powrotu i uporządkowaniu stosu. Dla małych funkcji narzut czasowy (odpowiednio: narzut w postaci dodatkowych rozkazów generowanego kodu) może w skrajnych przypadkach przewyższyć czas wykonania zasadniczej treści funkcji (odpowiednio: rozmiar wygenerowanego kodu realizującego treść funkcji).

Alternatywą dla takiego wywoływania funkcji jest ich bezpośrednie wstawianie (w miejscu wywołania) w generowany kod. Z punktu widzenia C++ wywołania funkcji wstawialnych nie różnią się niczym od wywołań „zwykłych" funkcji, dla kompilatora natomiast funkcje wstawialne stanowią dodatkową okazję do optymalizacji.

Ujemną stroną funkcji wstawialnych jest dodatkowy (zazwyczaj) narzut na rozmiar kodu wynikowego, bowiem wywołanie funkcji (w kodzie źródłowym) jest teraz zastępowane nie standardową sekwencją wywołującą, lecz *kompletną treścią* wywoływanej funkcji — notabene znowu spotykamy się ze zjawiskiem przyspieszenia wykonywania kodu za cenę dodatkowej pamięci. Ponadto, jak przed chwilą wspominaliśmy, pętle obejmujące niewielkie fragmenty kodu są o tyle cenne z punktu widzenia efektywności, iż często mogą w całości mieścić się w pamięci podręcznej; dodatkowe narzuty na rozmiar generowanego kodu mogą taką szansę zaprzepaścić.

Funkcja traktowana jest przez kompilator jako wstawialna, jeżeli opatrzona jest klauzulą `inline`; odnosi się to również do metod definiowanych poza ciałem klasy. Metody definiowane w ciele klasy traktowane są automatycznie jako wstawialne i nie wymagają klauzuli `inline`. Wyjaśnia to dokładniej przykład z wydruku 4.8.

Wydruk 4.8.
Przykłady funkcji wstawialnych

```
class TMyClass
{
private:
    int FValue;
public:
    void SetValue(int NewValue) { FValue = NewValue; }
    int  GetValue() const { return(FValue); }
    void DoubleIt();
    void HalveIt();
};
```

```
inline void TMyClass::DoubleIt()
{
    FValue *= 2;
}

void TMyClass::HalveIt()
{
    FValue /= 2;
}

inline int Negate(int InitValue)
{
    return(-InitValue);
}

void SomeFunction()
{
TMyClass Abc;
int NegNewVal;

    Abc.SetValue(10);
    Abc.DoubleIt();
    NegNewVal = Negate(Abc.GetValue());
}
```

Funkcje: `TMyClass::SetValue()`, `TMyClass::GetValue()`, `TMyClass::DoubleIt()` i `TMyClass::Negate()` są funkcjami wstawialnymi, w przeciwieństwie do funkcji `TMyClass::HalveIt()` i `SomeFunction()`.

Funkcja opatrzona klauzulą `inline` nie zostanie jednak zrealizowana jako funkcja wstawialna, jeżeli nie będzie zdefiniowana przed pierwszym wywołaniem (w sensie sekwencyjnego tekstu kodu źródłowego) — jeżeli na wydruku 4.8 funkcja `Negate()` zostałaby zdefiniowana po funkcji `SomeFunction()`, jej wywołanie zostałoby zrealizowane przy użyciu standardowej sekwencji wywołującej. Kompilator ignoruje ponadto dyrektywę `inline` w przypadku funkcji zawierającej specyfikację wyjątków, używającej jako parametrów obiektów przekazywanych przez wartość lub zwracającej jako wynik obiekt posiadający destruktor.

Funkcje wstawialne stanowią pewien problem w przypadku śledzenia aplikacji — nie ma np. sposobu na „zastawienie" punktu przerwania w ciele funkcji. Można więc nakazać kompilatorowi ignorowanie klauzuli `inline` i realizację wywołań opatrzonych nią funkcji na równi z wywołaniami zwykłych funkcji — należy w tym celu zaznaczyć opcję *Disable inline expansions* w sekcji *Debugging* na karcie *Compiler* opcji projektu.

W odniesieniu do naszej przykładowej aplikacji stwierdziliśmy metodą prób i błędów, iż jedyną funkcją wartą opatrzenia jej klauzulą `inline` jest funkcja `CanPlace()` — co daje kolejne przyspieszenie:

obecny czas wykonania: 23,46 sekundy;

usprawnienie w tym kroku: 0,9 proc.;

przyspieszenie globalne: 3391 razy.

Eliminowanie zmiennych i obiektów tymczasowych

Tworzenie i likwidacja zmiennych — zwłaszcza obiektów — tymczasowych zajmuje trochę czasu, należy więc ograniczać ich użycie. Tymczasowe łańcuchy, obiekty i zmienne proste powinny być deklarowane w jak najmniejszym możliwym zakresie. Spójrzmy na wydruk 4.9:

Wydruk 4.9.
Nieoptymalne korzystanie ze zmiennych tymczasowych

```
void SomeFunc(bool State)
{
    TComplexObj Obj;
    String Name;
    int Length;

    if (State) {
        Name = "To ja, łańcuch.";
        Length = Name.Length();
        DoSomething(Name, Length);
    } else {
        Obj.Weight = 5.2;
        Obj.Speed = 0;
        Obj.Acceleration = 1.5;
        Obj.DisplayForce();
    }
}
```

Trzy tworzone obiekty tymczasowe — `Obj`, `Name` i `Length` — nigdy nie są potrzebne jednocześnie: zależnie od wartości zmiennej `state` wykorzystywany jest tylko pierwszy albo tylko dwa ostatnie z nich. Należy więc opóźnić ich tworzenie do czasu, kiedy rzeczywiście okażą się potrzebne, co ilustruje wydruk 4.10.

Wydruk 4.10.
Właściwe umieszczenie deklaracji obiektów tymczasowych

```
void SomeFunc(bool State)
{
    if (State) {
        String Name;
        Name = "To ja, łańcuch.";
        DoSomething(Name, Name.Length());
    } else {
        TComplexObj Obj;
        Obj.Weight = 5.2;
        Obj.Speed = 0;
        Obj.Acceleration = 1.5;
        Obj.DisplayForce();
    }
}
```

Możliwe jest dalsze zwiększenie efektywności wykorzystania obiektów tymczasowych poprzez łączenie ich deklaracji z inicjalizacją za pomocą konstruktora kopiującego — czego przykład przedstawia wydruk 4.11.

Wydruk 4.11.
Automatyczne inicjalizowanie obiektów tymczasowych

```
void SomeFunc(bool State)
{
    if (State) {
        String Name("To ja, łańcuch.");
        DoSomething(Name, Name.Length());
    } else {
        TComplexObj Obj(5.2, 0, 1.5);
        Obj.DisplayForce();
    }
}
```

Używanie *przyrostkowych* (postfiksowych) operatorów inkrementacji i dekrementacji zawsze wiąże się z tworzeniem tymczasowej zmiennej pamiętającej poprzednią wartość obiektu. Spójrzmy na wydruk 4.12:

Wydruk 4.12.
Nieoptymalność wynikająca z użycia operatora przyrostkowego

```
class TMyIntClass
{
private:
    int FValue;
public:
    TMyIntClass(int Init) { FValue = Init; }
    int GetValue() { return FValue; }
    // przyrostkowy operator ++
    int operator++(int) {
        int Tmp = FValue;
        FValue = FValue + 1;
        return(Tmp);
    }
    // przedrostkowy operator ++
    int operator++() {
        FValue = FValue + 1;
        return(FValue);
    }
};


void MyFunc()
{
TMyIntClass A(1);

    A++; // wartością tego wyrażenia jest wartość obiektu A sprzed
         // inkrementacji, ukrywająca się pod tymczasową zmienną lokalną tmp

    ++A; // wartością tego wyrażenia jest wartość obiektu po dekrementacji,
         // dzięki czemu nie jest potrzebna zmienna przechowująca jego
         // poprzednią wartość
}
```

Jak wyjaśniają to komentarze, z inkrementacją przyrostkową wiążą się dwie różne wartości: ta reprezentująca wynik wyrażenia i ta powstająca w wyniku inkrementacji.

W klasie `TMyIntClass` wartość obiektu reprezentowana jest przez prywatne pole `FValue`. Po jego zwiększeniu za pomocą operatora przyrostkowego musi być jeszcze dostępna jego wartość sprzed inkrementacji, by można ją było zwrócić w instrukcji `return` jako „wynik" operatora — i temu właśnie celowi służy zmienna `tmp`. W przypadku inkrementacji przedrostkowej poprzednia wartość obiektu nie jest już istotna, nie potrzeba więc wspomnianej zmiennej tymczasowej.

W przypadku przekazywania obiektu przez wartość (jako parametr funkcji) tworzona jest (za pomocą konstruktora kopiującego) jego tymczasowa kopia, reprezentująca go w treści funkcji. Kopia taka nie jest jednak tworzona, jeżeli przekazać obiekt przez referencję i poprzedzić ją klauzulą `const` — kopia obiektu jest wówczas niepotrzebna, bowiem funkcja nie dokonuje w nim żadnych zmian (nie pozwoliłby na to kompilator). Tak więc np. funkcja zadeklarowana jako:

```
MyFunc(const TMyClass &A)
```

jest z opisanych powodów bardziej efektywna od funkcji zadeklarowanej jako:

```
MyFunc(TMyClass A)
```

choć należy przyznać, iż możliwość takiej optymalizacji zależy oczywiście od treści funkcji. Nawet jednak w tak zoptymalizowanym przypadku tworzona będzie tymczasowa kopia obiektu, jeżeli typ obiektu użytego w wywołaniu nie jest dokładnie taki sam, jak typ zadeklarowanego parametru formalnego.

W przypadku, gdy wynikiem funkcji jest obiekt, o wiele efektywniejsze jest tworzenie jego instancji „w miejscu", w bezpośrednim związku z instrukcją `return`, niż wykorzystywanie w charakterze wartości zwrotnej jakiejś odrębnej zmiennej tymczasowej. Zasadę tę ilustruje wydruk 4.13.

Wydruk 4.13.
Optymalizacja funkcji zwracającej jako wynik instancję obiektu

```
TMyClass TmpReturnFunc(bool Something)
{
    TMyClass TmpObj(0, 0);

    if (Something) {
        TMyClass Obj1(1.5, 2.2),
                 Obj2(1.8, 6.1);
        TmpObj = Obj1 + Obj2;
    }
    return(TmpObj);
}

TMyClass FastReturnFunc(bool Something)
{
    if (Something) {
        TMyClass Obj1(1.5, 2.2),
                 Obj2(1.8, 6.1);
        return(Obj1 + Obj2);
    } else {
        return(TMyClass(0, 0));
    }
}
```

```
void MyFunc()
{
    TMyClass A;

    A = TmpReturnFunc(AddThem);   // Obiekt tymczasowy zwrócony przez TmpReturnFunc()
    A = FastReturnFunc(AddThem);  // Obiekt zwrócony bezpośrednio w zmiennej A
}
```

W przypadku funkcji `TmpReturnFunc()` zmienna tymczasowa `TmpObj` jest konstruowana i, po przepisaniu jej wartości do wynikowej instancji obiektu, niszczona przez destruktor. W przypadku funkcji `FastReturnFunc()` wynikowa instancja jest od razu inicjowana żądaną wartością, bez używania pomocniczych zmiennych tymczasowych.

Jednokrotne obliczanie wyrażeń o niezmieniającej się wartości

Powtarzające się wielokrotnie w pętli wartościowanie wyrażenia, dającego za każdym razem ten sam wynik, może być wyniesione przed pętlę. W przypadku wyrażenia warunkowego może to niekiedy oznaczać podzielenie jednej pętli na dwie odrębne, jak ilustruje to wydruk 4.14.

Wydruk 4.14.
Wynoszenie przed pętle wyrażeń o niezmieniającej się wartości

```
// Ta pętla wykonuje się powoli.
for (int i = 0 ; i < 10 ; i++) {
    if (InitializeType == Clear) {
        a[i] = 0;
        b[i] = 0;
    } else {
        a[i] = y[i];
        b[i] = x + 5;
    }
}

// Te pętle wykonają się szybciej.
if (InitializeType == Clear) {
    for (int i = 0 ; i < 10 ; i++) {
        a[i] = 0;
        b[i] = 0;
    }
} else {
    int Total = x + 5;
    for (int i = 0, Total = x + 5 ; i < 10 ; i++) {
        a[i] = y[i];
        b[i] = Total;
    }
}
```

Indeksowanie tablic i arytmetyka wskaźników

W przypadku wyrażenia zawierającego skomplikowane indeksowanie tablic i (lub) złożone obliczenia z użyciem wskaźników, o wiele efektywniejsze jest posługiwanie się wskaźnikami do obiektów docelowych.

Wydruk 4.15 przedstawia fragment kodu naszej przykładowej aplikacji, który naprawdę trudno posądzić o prostotę:

Wydruk 4.15.
Skomplikowane indeksowanie tablic

```
// skomplikowane indeksowanie tablic w ramach funkcji SolveStandardWord()
k = CrozLetterPos[Words[CurrWordNum].Letters[CurrLetterIdx]].NumPositions-1;
while (k >= 0) {
   CurrX = CrozLetterPos[Words[CurrWordNum].Letters[CurrLetterIdx]].Position[k].x;
   CurrY = CrozLetterPos[Words[CurrWordNum].Letters[CurrLetterIdx]].Position[k].y;
...
}

.....

// skomplikowane indeksowanie tablic w ramach funkcji PlaceWord()

CrozLetterPos[Words[WordNum].Letters[LetterIdx]].Position[CrozLetterPos[Words[WordNum]
.Letters[LetterIdx]].NumPositions].x    = CurrX;

CrozLetterPos[Words[WordNum].Letters[LetterIdx]].Position[CrozLetterPos[Words[WordNum]
.Letters[LetterIdx]].NumPositions].y    = StartY;

CrozLetterPos[Words[WordNum].Letters[LetterIdx]].NumPositions++;
```

Mamy tu do czynienia nie tyle z niezmienną wartością wyrażenia sensu stricte, lecz wielokrotnym, skomplikowanym określaniem tego samego miejsca w pamięci — jest nim ustalony element tablicy `CrozLetterPos[]`, będący strukturą typu `TCrozLetterPos`, którego desygnatorem jest złożone, wielokrotnie obliczane, wyrażenie indeksowe. Wydruk 4.16 pokazuje, jak można rzecz całą uprościć, zastępując ów desygnator wskaźnikiem.

Wydruk 4.16.
Zoptymalizowane odwołanie do ustalonego elementu tablicy

```
// fragment zoptymalizowanej funkcji SolveStandardWord()
   ...
TCrozLetterPos *TmpPos;
   ...
TmpPos = &CrozLetterPos[Words[CurrWordNum].Letters[CurrLetterIdx]];
k = TmpPos->NumPositions-1;
while (k >= 0 && TmpPos->PlacedLetter[k].WordPlacedIdx >= PlacedIdxLimit) {
   CurrX = TmpPos->PlacedLetter[k].Position.x;
   CurrY = TmpPos->PlacedLetter[k].Position.y;
   ...
}
// fragment zoptymalizowanej funkcji PlaceWord()

  ...

TCrozLetterPos *TmpPos;
  ...
TmpPos = &CrozLetterPos[Words[WordNum].Letters[LetterIdx]];
TmpPos->PlacedLetter[TmpPos->NumPositions].Position.x    = CurrX;
TmpPos->PlacedLetter[TmpPos->NumPositions].Position.y    = StartY;
```

```
TmpPos->PlacedLetter[TmpPos->NumPositions].WordPlacedIdx =
CrozGrid[StartY][CurrX].HorizPlacedIdx;
TmpPos->NumPositions++;

  ...
```

Poza niezaprzeczalną poprawą czytelności kodu zmiana ta przyniosła również pewną korzyść pod względem czasu wykonania programu:

obecny czas wykonania: 22,28 sekundy;

usprawnienie w tym kroku: 5,3 proc.;

przyspieszenie globalne: 3571 razy.

Arytmetyka zmiennoprzecinkowa

Układy arytmetyki zmiennoprzecinkowej stanowią integralną część współczesnych procesorów, same zaś obliczenia zmiennoprzecinkowe wykonywane są niezwykle efektywnie — przykładowo funkcje przestępne i trygonometryczne, obliczane niegdyś wyłącznie w sposób programowy, dziś dostępne są za pośrednictwem jednego lub kilku rozkazów procesora. Mimo to C++Builder oferuje dodatkowe środki dla dodatkowego zwiększenia efektywności obliczeń zmiennoprzecinkowych — jest nim moduł *fastmath.cpp* i stowarzyszony z nim plik nagłówkowy *fastmath.h*. Moduł ten zawiera efektywne implementacje ok. 50 funkcji trygonometrycznych; nazwy tych funkcji rozpoczynają się od przedrostka `_fm_`. Większość standardowych funkcji zmiennoprzecinkowych — `cos()`, `exp()`, `log()`, `sin()`, `atan()`, `sqrt()` itp. — jest standardowo „mapowana" do odpowiednich funkcji `_fm_`*xxxx*`()`. Aby wymusić zaprzestanie tego mapowania, należy umieścić dyrektywę `#define _FM_NO_REMAP` przed dyrektywą `#include`, włączającą plik `<fastmath.h>`. Funkcje modułu `fastmath` będą wówczas dostępne jedynie przez swe oryginalne nazwy, rozpoczynające się od `_fm_`.

Efektywność funkcji modułu `fastmath` bierze się m.in. stąd, iż nie sprawdzają one i nie sygnalizują większości błędnych sytuacji; stąd wniosek, iż na ich użycie można sobie pozwolić tylko w aplikacji dobrze przetestowanej, gdzie w dodatku szybkość jest czynnikiem krytycznym.

Pewna dodatkowa możliwość drobnej optymalizacji operacji zmiennoprzecinkowych polega na zastępowaniu dzielenia nieco szybszym mnożeniem. Jeżeli mianowicie w pętli wielokrotnie powtarzane są obliczenia w rodzaju A/B, korzystniej będzie obliczyć (przed pętlą) wartość T = 1/B i zamiast wspomnianego ilorazu obliczać iloczyn A*T.

Niestety, w większości przypadków zabieg taki może przynieść dodatkową stratę dokładności obliczeń. Stanie się tak w przypadku, gdy wartość B posiada skończoną reprezentację bitową (w ramach przyjętej reprezentacji liczb zmiennoprzecinkowych), zaś wartość 1/B reprezentacji takiej nie posiada. Takimi liczbami są np. 3,0 lub 10,0 — zarówno 1/3, jak i 1/10 są w reprezentacji binarnej ułamkami okresowymi.

Inne możliwości optymalizowania kodu w C++

Spośród innych, bardziej specjalizowanych możliwości wysokopoziomowej optymalizacji kodu na uwagę zasługują trzy następujące:

- należy unikać odwołania do mechanizmów RTTI — nie da się tego jednak zrobić w aplikacjach korzystających z rzutowania dynamicznego (`dynamic_cast`) lub używających operatora `typeid`;
- funkcje wirtualne są mniej efektywne od statycznych, ponieważ ich wywołanie wiąże się z przeglądaniem V-tablic;
- operacje graficzne są operacjami czasochłonnymi w przełożeniu na czas procesora, a więc dążenie do szybkości programu powinno iść w parze z ich minimalizacją. Należy więc ograniczać (o ile to możliwe) obszary „odświeżania" grafiki tylko do tych fragmentów, które faktycznie tego wymagają, wywołując funkcję `InvalidateRect()` z odpowiednimi parametrami. Należy również unikać nieprzyjemnego migotania obrazu, konstruując właściwą jego zawartość w roboczej bitmapie i kopiując tę ostatnią na ekran, bądź też dokonując czasowego ukrywania obrazu za pomocą funkcji `Hide()` i `Show()`.

Aby aplikacja zdolna była w ogóle reagować na polecenia użytkownika, musi mieć czas na bieżące przetwarzania komunikatów generowanych przez Windows w wyniku np. naciskania klawiszy czy poruszania myszą. W tym celu wszelkie czasochłonne obliczenia powinny być przeplatane periodycznymi wywołaniami funkcji `Application->ProcessMessages()` — i tak właśnie robimy w naszej aplikacji. „Wykomentowanie" wiersza, zawierającego wspomniane wywołanie, przyspiesza co prawda działanie programu średnio o 1,5 procent, lecz jednocześnie użytkownik traci możliwość jakiejkolwiek interakcji z aplikacją — nie działają przyciski, opcje menu ani nawet standardowe zamknięcie okna.

Techniki optymalizacji danych

Podobnie jak w zakresie kodu, tak i w zakresie danych programu kryją się pewne możliwości jego przyspieszenia. Jak już pisaliśmy, procesor posiada pamięć podręczną nie tylko dla kodu, lecz również dla danych; sekwencyjny dostęp do pamięci jest więc szybszy od dostępu nieregularnego, rozproszonego, dane są bowiem przenoszone do pamięci podręcznej całymi blokami i przy sekwencyjnym ich przetwarzaniu większość odczytów i zapisów wykonuje się w pamięci podręcznej, nie w pamięci RAM. W przełożeniu na język C++ oznacza to, iż „klasyczne" tablice są jako struktury danych efektywniejsze w obsłudze od np. list łączonych, drzew czy tablic rozproszonych, które posługują się wyrywkowo przydzielanymi, nieprzylegającymi do siebie obszarami pamięci.

Można jednak zmniejszyć tę niekorzystną tendencję wyrażoną w ostatnim zdaniu, przydzielając a priori większy, spójny fragment pamięci i w ramach niego dokonywać subalokacji węzłów listy czy drzewa. Powinno to zdecydowanie zmniejszyć opisaną „fragmentację" przydziału.

Ponadto używanie mniejszych (rozmiarowo) struktur danych powoduje, iż większa ich ilość ma szansę zmieścić się jednocześnie w pamięci podręcznej. Dotyczy to zwłaszcza elementów dużych tablic.

Nie bez znaczenia jest również sposób aranżacji używanych danych. Jeżeli przykładowo częścią danych naszego programu są dwa wektory `X A[]` i `Y B[]`, to sposób ich implementacji powinien zależeć od tego, czy będą one wykorzystywane łącznie, czy też niezależnie od siebie. W pierwszym przypadku najodpowiedniejsza byłaby implementacja „przeplatana", w której elementy obydwu wektorów występują na przemian, tworząc w efekcie tablicę rekordów; w drugim przypadku lepsze będą dwie niezależne tablice. Ideę tę ilustruje wydruk 4.17.

Wydruk 4.17.
Dwa sposoby implementacji dwóch wektorów

```
// implementacja "przeplatana"

struct {
    X A;
    Y B;
} T[100];

void SimFunc()
{
    int AveDiff = 0;
    for (int i = 0 ; i < 100 ; i++) {
        AveDiff += T.B[i] - T.A[i];
    }
    AveDiff /= 100;
}

// implementacja niezależna
X A[100];
Y B[100];

void SepFunc()
{
    int Sum = 0;
    for (int i = 0 ; i < 100 ; i++) {
        Sum += A[i];
    }

    ...

    B[Index] = Q * R;
}
```

Zastosowaliśmy tę ideę do naszej aplikacji, tworząc struktury `TCrozLetterPos` i `TUnsolved`; efekt był nieco zaskakujący — czas wykonania aplikacji *zwiększył* się do 22,65 sekundy. Przyczyna tego stanu rzeczy stała się jasna już po chwili — po prostu naszymi działaniami utrudniliśmy kompilatorowi optymalizację odwołań do tablic.

Otóż w wyniku optymalizacji struktury `TCrozLetterPos` utworzona została 12-bajtowa struktura `TPlacedLetter`, stanowiąca element tablicy. Jak wiadomo, dostęp do elemen-

tów tablicy związany jest m.in. z operacją mnożenia przez rozmiar pojedynczego elementu, zaś spośród mnożeń najszybsze są mnożenia przez potęgę dwójki, dają się one bowiem zrealizować za pomocą tylko przesunięć bitowych — co skwapliwie wykorzystuje kompilator, znając przecież rozmiar pojedynczego elementu każdej z tablic. Jak wiadomo, 12 nie jest potęgą dwójki, a więc dostęp do tablicy `TPlacedLetter[]` odbywał się przy pomocy zwyczajnych mnożeń.

Wskazane więc było sztuczne „rozepchanie" struktury `TPlacedLetter` do wielkości 16 bajtów, przez dodanie do niej fikcyjnego pola 4-bajtowego; my postąpiliśmy nieco oszczędniej, zmieniając typ dwóch pól struktury `TPos` z `int` na `short`, dzięki czemu struktura `TPlacedLetter` zmniejszyła się do rozmiaru 8 bajtów. Ponieważ jednocześnie w wyniku tego zabiegu struktura `TAdjLetter` o dotychczasowym, optymalnym rozmiarze 8 bajtów skurczyła się teraz do 6 bajtów, konieczne było jej sztuczne powiększenie do poprzedniego rozmiaru, co uczyniliśmy za pomocą fikcyjnego pola typu `short`. W efekcie, zamiast pogorszenia efektywności, otrzymaliśmy nieznaczną poprawę:

obecny czas wykonania: 21,79 sekundy;

usprawnienie w tym kroku: 2,2 proc.;

przyspieszenie globalne: 3651 razy.

W przypadku tablic wielowymiarowych kolejność poszczególnych wymiarów powinna wynikać z kolejności przetwarzania elementów tablicy przez aplikację — należy mianowicie dążyć do tego, by w przełożeniu na rodzaj dostępu do pamięci przetwarzanie to stanowiło dostęp sekwencyjny. Jak wiadomo, elementy tablicy ułożone są w pamięci w ten sposób, iż wartość skrajnego prawego indeksu zmienia się najszybciej — i tak np. dla tablicy `T[5][10]` układ jej elementów w pamięci jest następujący:

```
T[0][0]
T[0][1]
T[0][2]

...

T[0][9]
T[1][0]
T[1][1]

...

T[4][0]

...

T[4][8]
T[4][9]
```

Znając z grubsza kolejność przetwarzania elementów tablicy, możemy przełożyć ją na odpowiednią kolejność wymiarów — albo dysponując konkretną deklaracją tablicy możemy w sposób optymalny ustalić kolejność przetwarzania jej elementów, o ile oczywiście pozwalają na to warunki aplikacji. Dwa fragmenty kodu na wydruku 4.18 stanowią przykład drugiego podejścia.

Wydruk 4.18.
Kolejność przetwarzania elementów tablicy wielowymiarowej

```
// kolejność nieoptymalna – elementy nie są przetwarzane sekwencyjnie
for (a = 0 ; a < 10 ; a++) {
    for (b = 0 ; b < 5 ; b++) {
        T[b][a] = 0;
    }
}

// kolejność optymalna – elementy są przetwarzane zgodnie z ułożeniem
// w pamięci
for (b = 0 ; b < 5 ; b++) {
    for (a = 0 ; a < 10 ; a++) {
        T[b][a] = 0;
    }
}
```

Kolejną techniką optymalizacji danych, z której notabene szeroko korzysta Win 32 API, jest współdzielenie pojedynczego egzemplarza obiektu zamiast posługiwania się oddzielnymi egzemplarzami danej klasy. Taki współdzielony obiekt musi być kontrolowany za pomocą tzw. licznika odwołań (ang. *reference counter*), kontrolującego jego wykorzystanie; licznik ten inicjowany jest wartością 0 podczas kompilacji lub przy rozpoczęciu wykonywania programu.

Odpowiedzią na żądanie utworzenia obiektu w sytuacji, gdy wspomniany licznik ma wartość 0, jest faktyczne utworzenie egzemplarza obiektu, zwrócenie jego wskaźnika i nadanie licznikowi wartości 1. Każde kolejne żądanie będzie jednak powodowało wyłącznie inkrementację licznika i zwracanie wskaźnika do *istniejącego* egzemplarza. Z kolei każda „destrukcja" obiektu będzie powodować jedynie dekrementację licznika i dopiero wówczas, gdy wynikiem tej dekrementacji będzie wartość zerowa, dokona się faktyczna destrukcja fizycznego egzemplarza obiektu.

I wreszcie — niepotrzebne dane, stanowiące pozostałość po poprzednich wersjach, są równie niepożądane, jak niepotrzebne fragmenty kodu.

Programowanie na poziomie asemblera

W dzisiejszych czasach programowanie na poziomie asemblera bywa często postrzegane jako coś na kształt czarnej magii. Nic w tym dziwnego, wszak repertuar rozkazów procesora Pentium II obejmuje ponad 200 instrukcji operujących na liczbach całkowitych, prawie 100 instrukcji zmiennoprzecinkowych i ok. 30 instrukcji systemowych; należy dodać do tego ok. 60 instrukcji strumieniowych (ang. *SIMD — Single-Instruction-Multiple-Data*) oraz tyleż samo instrukcji MMX. Całości dopełniają wszelkie subtelności architektury, wpływające bezpośrednio na efektywność realizacji rozkazów — pamięci podręczne 1. i 2. poziomu, wielokrotne strumienie instrukcji umożliwiające równoległe wykonywanie instrukcji, przewidywanie rozgałęzień, przemianowywanie rejestrów itp.

Opanowanie tej złożoności daje jednak bardzo wyraźną korzyść — mianowicie pełną kontrolę nad działaniami wykonywanymi przez procesor na rzecz danej aplikacji. Kod

asemblerowy kształtowany jest przez programistę na podstawie koncepcji zrodzonej w jego umyśle, co jest czynnością jakościowo różną od optymalizacji automatycznej, wykonywanej przez kompilator na podstawie fragmentów kodu źródłowego.

Przedstawienie chociażby tylko zarysu programowania w języku asemblera (z uwzględnieniem specyfiki konkretnych procesorów) wykracza poza ramy tej książki. Zainteresowanych czytelników odsyłamy do następującej literatury, zalecanej przez Autorów oryginału:

- *The Art of Assembly Language Programming* (*http://webster.cs.ucr.edu*) — ponadtysiącpięćsetstronicowy podręcznik w formacie `.PDF` (do ściągnięcia) lub HTML (do selektywnego przeglądania *online*);
- *Intel Architecture Software Developer's Manual* (*http://developer.intel.com/design/processor/*) — wszystko na temat procesorów Pentium, do ściągnięcia podręczniki dotyczące poszczególnych odmian procesora;
- Iczelion Win32 Assembly HomePage (*http:// win32asm.cjb.net*) — strona poświęcona programowaniu w asemblerze w środowisku Win32 z wieloma tutorialami i odsyłaczami;
- *Optimizing for Pentium Microprocessors* (*http://www.agner.ord/assem/*) — strona zawierająca wskazówki odnośnie optymalizacji kodu asemblerowego pod kątem procesorów Pentium;
- grupa dyskusyjna *comp.lang.asm.x86* poświęcona programowaniu procesorów serii x86.

Z wydawnictw w języku polskim polecamy między innymi:

- P. Metzger, A. Jełowicki *Anatomia PC*, wyd. V, Helion 1999 — oprócz wielu szczegółów konstrukcyjnych komputerów zawiera opisy (wraz ze szczegółowymi danymi) nowoczesnych mechanizmów przyspieszających wykonanie programów przez procesory serii Pentium oraz konkurencyjne — AMD, Cyrix, IDT i Rise.

Programista zainteresowany postacią kodu generowanego przez kompilator może ów kod obejrzeć w okienku CPU zintegrowanego debuggera. Należy w tym celu ustawić punkt przerwania (*breakpoint*) na interesującym wierszu kodu źródłowego i uruchomić aplikację, doprowadzając do zatrzymania wykonania w tym punkcie; następnie należy wyświetlić okno ukazujące przekład (na język asemblera) instrukcji kodu źródłowego z otoczenia instrukcji objętej punktem przerwania — służy do tego opcja *View|Debug Windows|CPU*. Za pomocą sąsiedniej opcji (FPU) można także zobaczyć zawartość rejestrów, stosu i znaczników jednostki zmiennoprzecinkowej.

Można także uzyskać czytelną postać przekładu w bardziej trwałej postaci. Należy mianowicie dodać znacznik `–B` w sekcji *CFLAG1* pliku głównego projektu (dostępnego za pośrednictwem opcji *Project|Edit Option Source*) i zaznaczyć na karcie *Tasm* opcji projektu pozycje *Generate wydruk* i *Expanded wydruk* — w efekcie dla każdego z plików **.cpp*, wchodzących w skład projektu, wygenerowany zostanie równoważny plik `*.asm`, odzwierciedlający przekład dokonany przez kompilator. Ten sam efekt osiągnąć można, uruchamiając kompilator C++Buildera z wiersza poleceń w następujący sposób:

```
BCC32 –S moduł.cpp
```

gdzie *moduł* jest nazwą modułu źródłowego; może okazać się konieczne użycie opcji –I, wskazującej ścieżkę dostępu do dołączanych plików. Wygenerowany plik *.asm* zawierał będzie (oprócz wygenerowanego kodu) „wykomentowane" średnikami instrukcje kodu źródłowego.

Włączanie (do treści funkcji) kodu w języku asemblera odbywa się za pomocą słowa kluczowego asm, po którym następuje blok instrukcji asemblerowych; poza instrukcją asm możliwy jest także dostęp do rejestrów procesora — ich symbole należy poprzedzić podkreśleniem; ma to jednak sens jedynie w bezpośrednim sąsiedztwie instrukcji asm. Wydruk 4.19 prezentuje napisaną w asemblerze funkcję obliczającą wartość silni argumentu typu int:

Wydruk 4.19.
Obliczanie funkcji „silnia"

```
int Factorial(int Value)
{
   _EDX = Value;
    asm {
        push ebp
        mov  ebp,esp
        push ecx
        push edx
        mov [ebp-0x04],edx
        mov eax,0x00000001
        cmp dword ptr [ebp-0x04],0x02
        jl end
top:
        imul dword ptr [ebp-0x04]
        dec dword ptr [ebp-0x04]
        cmp dword ptr [ebp-0x04],0x02
        jnl top
end:
        pop edx
        pop ecx
        pop ebp
    }
    return(_EAX);
}
```

Tę funkcję można było napisać cokolwiek efektywniej — wersja prezentowana poniżej obywa się bez zmiennych pomocniczych w pamięci, wykorzystując jedynie rejestry EAX, ECX i EDX. Ponadto w przeciwieństwie do prezentowanego wyżej pierwowzoru poniższa wersja obsługuje poprawnie sytuacje wyjątkowe — dla argumentu ujemnego albo zbyt dużego, by jego silnia zmieściła się w zakresie typu int, wynikiem funkcji jest zero. O poprawności jej działania można się przekonać, uruchamiając projekt *AsmFactorial.bpr* z załączonej płyty CD-ROM.

```
int factorial(int val)
// (C) 2001 A.Grażyński
{
  _ECX = val;
  asm
```

```
  {
    xor eax,eax
    inc eax
// w EAX znajduje się domyślna wartość 1 dla argumentu 0

    test ecx,ecx
    jl  @@invalid // błąd, gdy argument ujemny
    jz  @@done    // gotowe, gdy argument zerowy

@@again:
    mul  ecx  // mnożenie EAX*ECX, wynik w EDX:EAX
    jc   @@ovfl
// jeżeli wynik mnożenia nie mieści się w EDX:EAX,
// ustawiany jest znacznik CF

    loop @@again

// jeżeli wynik nie mieści się w całości w EAX,
// to oznacza, że argument jest zbyt duży
    test edx,edx
    jnz  @@ovfl

// jeżeli EAX jest ujemne, to jest to wynik nadmiaru
// (argument jest za duży)
    test eax,eax
    jns  @@done
@@ovfl:
// Tutaj sterowanie trafia, gdy wartość argumentu jest zbyt duża,
// by jego silnia zmieściła się w zakresie typu int.
// Niniejsza funkcja sygnalizuje ten fakt, zwracając wartość zero.
// Użytkownik może zaprogramować w tym miejscu swoją własną obsługę.

@@invalid:
    xor eax,eax
@@done:
  }
  return(_EAX);
}
```

Jeżeli opatrzymy tę funkcję klauzulą `__fastcall`, będzie ona oczekiwała argumentu w rejestrze `EAX` i w tymże rejestrze oczekiwany będzie wynik przez funkcję wywołującą. Dotychczasowe instrukcje poza blokiem `asm` nie będą więc potrzebne (dla przejrzystości usunąłem komentarze):

```
int __fastcall factorial(int val)
// (C) 2001 A.Grażyński
{
  asm
  {
    push  ecx
    mov ecx,eax
    xor eax,eax
    inc eax
    test ecx,ecx
    jl  @@invalid
    jz  @@done
@@again:
    mul  ecx
```

```
    jc   @@ovfl
    loop @@again
    test edx,edx
    jnz  @@ovfl
    test eax,eax
    jns  @@done
@@ovfl:
// Tutaj sterowanie trafia, gdy wartość argumentu jest zbyt duża,
// by jego silnia zmieściła się w zakresie typu int.
// Niniejsza funkcja sygnalizuje ten fakt, zwracając wartość zero.
// Użytkownik może zaprogramować w tym miejscu swoją własną obsługę.

@@invalid:
    xor eax,eax
@@done:
    pop ecx
  }
}
```

Wykorzystując wstawki asemblerowe w regularnym kodzie C++, należy pamiętać o kilku istotnych sprawach:

- najwięcej korzyści osiąga się drogą programowania rzeczonych fragmentów kodu „od zera” — naśladowanie kodu wygenerowanego przez kompilator (w nadziei jego „ulepszenia”) stanowi jednocześnie powielanie niedoskonałości przekładu;
- kod w języku asemblera powinien być podzielony na niewielkie, dobrze zrozumiałe fragmenty i o wyraźnie zarysowanych zasadach współpracy z otaczającym kodem w C++;
- ze względu na nieoczywistość swego związku z „zasadniczym” kodem aplikacji kod asemblerowy powinien być dobrze udokumentowany;
- nie należy zakładać żadnych zależności pomiędzy odrębnymi blokami `asm` — C++Builder nie gwarantuje zachowania zawartości rejestrów na zewnątrz tych bloków;
- należy starać się wykorzystać zalety parowania instrukcji, szczególnie w przypadku instrukcji długo wykonywanych, np. funkcji trygonometrycznych i w ogóle instrukcji zmiennoprzecinkowych; niekiedy może okazać się korzystna zmiana kolejności instrukcji w stosunku do ich logicznej kolejności wynikającej z oryginalnego algorytmu;
- instrukcje CALL powinny być w miarę możliwości bilansowane z instrukcjami RET, dla lepszego wykorzystania bufora stosu powrotu (*RSB — Return Stack Buffer*);
- wyrównanie danych do granicy słów 32-bitowych skutkuje efektywniejszą współpracą procesora zarówno z pamięcią RAM, jak i pamięcią podręczną;
- zazwyczaj włączanie wstawek asemblerowych powoduje rezygnację kompilatora z optymalizacji kodu funkcji zawierającej te wstawki; co prawda kompilator C++Buildera zalicza się do nielicznych wyjątków pod tym względem, należy jednak w miarę możności sprawdzać, czy zastosowanie kodu asemblerowego nie daje skutków odwrotnych do oczekiwanych.

Ponadto każdy blok `asm` powinien zachowywać zawartość rejestrów: EDI, ESI, ESP, EBP i EBX, a w przypadku funkcji z modyfikatorem `__fastcall` również rejestru ECX.

Optymalizacja uwarunkowań zewnętrznych

Konkretna aplikacja może wykazywać zróżnicowane zachowanie w różnych warunkach zewnętrznych, których najważniejszym aspektem są konkretne dane wejściowe — wspominaliśmy już o np. tym, iż niektóre algorytmy sortowania wykazują silną zależność swej efektywności od stopnia uporządkowania danych podlegających sortowaniu i tendencję tę zaobserwować można w odniesieniu do większości typowych aplikacji.

Nasza aplikacja „krzyżówkowa" pracuje nieco efektywniej, gdy słowa w pliku `.crz` uporządkowane są w kolejności malejącej długości — spostrzeżenie to pozwoliło nam dokonać kolejnego przyspieszenia:

obecny czas wykonania: 17,94 sekundy;

usprawnienie w tym kroku: 21,5 proc.;

przyspieszenie globalne: 4435 razy.

Rozwiązania produkowane przez opisywaną aplikację charakteryzują się zróżnicowaną jakością w rozumieniu punktacji każdego z nich; jeżeli więc w założonym odcinku czasu możemy zoptymalizować *wydajność* aplikacji (w sensie *najlepszego* produkowanego w tym czasie rozwiązania) będzie to miało dla użytkownika taką samą wartość, jak zwiększenie *szybkości* wykonania (*szybciej* produkowane będą *wartościowe* rezultaty), mimo iż tak naprawdę zwiększenie takie w sensie dosłownym nie następuje. Dla naszego zbioru, zawierającego listę 115 słów, kolejne słowa pobierane są do rozwiązania w kolejności zależnej w dużym stopniu od ich uporządkowania w pliku `.crz`, więc opisane przed chwilą zwiększenie wydajności uzyskać można, umieszczając na początkowych pozycjach tej listy słowa zawierające dużą liczbę liter wysoko punktowanych; wymaga to pewnego kompromisu wobec dosyć istotnego kryterium wspomnianego przed chwilą uporządkowania słów według ich malejącej długości.

Ponadto ograniczenie obszaru działania aplikacji (do określonego prostokąta) z jednej strony zmniejsza liczbę słów, które dają się sensownie ułożyć, lecz jednocześnie zmniejsza liczbę analizowanych rozwiązań, może więc (chociaż wcale nie musi — tu już mamy do czynienia z prawdziwą loterią) szybciej wyprodukować bardziej wartościowe rozwiązanie.

Optymalizacja szybkości — wnioski końcowe

Dotychczas zaprezentowaliśmy kilka istotnych technik optymalizacji różnorodnych aspektów aplikacji — projektu, używanych algorytmów, kodu, danych i uwarunkowań zewnętrznych — otrzymując wcale pokaźne efekty: 17,94 sekundy (na kompletną analizę 11-wyrazowej listy) to naprawdę nie to samo, co 22,1 godziny! Mimo to osiągi te nie stanowią jeszcze granicy możliwej do uzyskania szybkości — aplikacja wciąż bowiem zawiera elementy, które można zoptymalizować, w szczególności:

- parametry funkcji — niektóre z nich można poprzedzić klauzulą `const`;
- osobne zmienne boolowskie (np. `IsValid`, `IsLetter` i `IsBlank`) reprezentujące łącznie pewną sytuację, która może być reprezentowana w postaci pojedynczego typu wyliczeniowego;
- wyszukiwanie słów — punktacja związana z kwadratem nie leżącym na skrzyżowaniu słów równa jest zeru, można by więc nadać jej wartość równą *zanegowanemu* numerowi słowa, do którego przynależy; kwadraty leżące na skrzyżowaniu słów w dalszym ciągu odróżniałyby się od pozostałych *dodatnią* punktacją, natomiast ustalanie związku konkretnych słów z konkretnymi kwadratami stałoby się znacznie ułatwione;
- kolejność pobierania słów z listy — obecnie jest ona identyczna z fizyczną kolejnością słów w liście i na każdym poziomie rekursji testy startują od pierwszego słowa, tymczasem można by ją nieco uelastycznić, np. zgodnie z algorytmem „karuzelowym” (ang. *round-robin*) rozpoczynając przeglądanie listy od którejś z jej pozycji wewnętrznych (innej na każdym poziomie rekursji) i oczywiście kończąc na pozycji poprzedzającej. Nie przyczyni się to co prawda do zwiększenia szybkości, lecz da potencjalnie większą szansę na szybsze uzyskanie bardziej wartościowego rozwiązania;
- obliczanie łącznej punktacji — mimo iż biorą w nim udział tylko kwadraty o dodatniej punktacji, przeglądane są *wszystkie* kwadraty diagramu; można zmienić to niekorzystne zjawisko, wprowadzając jakąś dodatkową strukturę, umożliwiającą iterowanie *wyłącznie* po kwadratach leżących na skrzyżowaniu słów, na przykład tablicę (z licznikiem) zawierającą (w swej początkowej części) numery tych kwadratów.

Zainteresowanemu czytelnikowi pozostawiamy jako ćwiczenie zaimplementowanie ww. usprawnień i sprawdzenie ich wpływu na szybkość kompletnej analizy listy 11-wyrazowej, jak i na wydajność częściowej analizy listy 115-wyrazowej, czyli wartość najlepszego rozwiązania uzyskanego w danej jednostce czasu (np. w ciągu godziny).

Optymalizacja innych aspektów aplikacji

Szybkość aplikacji jest bardzo istotnym, lecz bynajmniej nie jedynym elementem decydującym o jej jakości z punktu widzenia użytkownika; innymi ważnymi — a więc wartymi wysiłków optymalizacyjnych — elementami są m.in.: rozmiar pliku wynikowego, zapotrzebowanie na pamięć RAM, efektywność współpracy z pamięcią dyskową i wpływ na obciążenie sieci.

Optymalizacja rozmiaru modułu wynikowego

Mniejszy rozmiar modułu wynikowego oznacza łatwiejszą jego dystrybucję, a to ze względu na dwa oczywiste ograniczenia — ograniczoną pojemność nośników (dyskietek i płyt CD-ROM) oraz ograniczoną prędkość transmisji internetowej. Wielkość modułu wynikowego przekłada się także w dużym stopniu na zajętość pamięci (wirtualnej), co ma niebagatelne znaczenie w przypadku uruchamiania kilku aplikacji (lub kilku instancji danej aplikacji) na pojedynczym komputerze.

Najważniejszym czynnikiem zwiększającym rozmiar modułu wynikowego jest wyrównywanie poszczególnych jednostek danych (sekcja *Data alignment* karty *Advanced Compiler*) — opcja *Byte* oznacza najlepsze „upakowanie" danych (brak wyrównywania), lecz jednocześnie pewne zagrożenie dla szybkości wykonania; kolejne gradacje wyrównania (na granicy wielokrotności 2, 4 lub 8 bajtów, odpowiednio *Word*, *Double word* i *Quad word*) oznaczają (potencjalnie) większą szybkość, lecz jednocześnie (potencjalnie) większy rozmiar generowanego modułu. Podobną przeciwstawność (pod względem szybkości i rozmiaru modułu) wykazują także opcje z sekcji *Code Optimization* na karcie *Compiler* — wybranie opcji *Speed* skutkuje z reguły powiększeniem rozmiaru modułu; dość interesującym kompromisem może być zaznaczenie opcji *Selected* i (po kliknięciu przycisku *Optimizations*) zaznaczenie wszystkich opcji z wyjątkiem *Inline intristic functions*, co może w zauważalnym stopniu ograniczyć rozmiar generowanego kodu bez zauważalnego spadku efektywności.

Inną techniką ograniczania rozmiaru dystrybuowanej aplikacji jest jej *pakietowanie*, polegające na wygenerowaniu nie jednego monolitycznego modułu, lecz kilku tzw. pakietów; zazwyczaj w kolejnych wersjach aplikacji niektóre pakiety (lub ich większość) pozostają niezmienione i nie wymagają powtórnego przesyłania. Powodzenie tego zabiegu uwarunkowane jest właściwym, modularnym zaprojektowaniem aplikacji.

Pisaliśmy już wcześniej, iż za cenę dodatkowej pamięci uzyskać można przyspieszenie aplikacji, na przykład przechowując wyniki obliczeń pośrednich; zjawisko to ma charakter dualny — za cenę dodatkowych obliczeń można mianowicie uzyskać zmniejszenie „pamięciożerności" aplikacji, na przykład obliczając niezbędne wielkości każdorazowo, kiedy są potrzebne, bez ich przechowywania. To oczywiście klasyczna postać znanego kompromisu „czas — pamięć".

Na kompromis o podobnym charakterze napotykamy w przypadku obiektów graficznych — jest to kompromis pomiędzy rozmiarem obrazu a jego jakością. Gdy rozmiar modułu jest czynnikiem krytycznym, można rozważyć różne techniki kompresji skutkujące zróżnicowaną wielkością pliku wynikowego i jednocześnie zróżnicowaną utratą jakości — i tak np. dla skomplikowanej grafiki korzystne może okazać się skompresowanie obrazów do formatu *JPEG*, zaś prostej grafiki do formatów *GIF* lub *PNG*. Podobną zasadę sformułować można także w stosunku do plików dźwiękowych.

Nie należy wreszcie zapominać o kompresji ostatecznych plików wynikowych za pomocą popularnych archiwizatorów, np. ZIP lub RAR; większość programów instalacyjnych wykonuje kompresję o zróżnicowanej efektywności.

Optymalizacja innych czynników

Poza szybkością aplikacji i rozmiarem generowanego modułu wynikowego optymalizacji podlegać mogą również między innymi następujące jej aspekty:

- dostęp do pamięci dyskowych — dostęp sekwencyjny jest efektywniejszy od dostępu „nieregularnego" ze względu na mniejszy ruch głowic odczytująco-zapisujących. Ze względu na to, iż dane dyskowe transmitowane są w postaci całych sektorów, należy

unikać ich odczytywania (zapisywania) w małych porcjach, np. wielkości pojedynczych znaków;

- czas rozruchu aplikacji — duży wpływ na czas rozruchu aplikacji, rozumiany jako czas osiągnięcia przez nią pełnej zdolności do reagowania na polecenia użytkownika, ma umiejętne rozplanowanie dołączanych bibliotek DLL w zakresie adresów pamięci wirtualnej poprzez określenie ich tzw. bazowych adresów ładowania (opcja *Image base* na karcie *Linker*)[8]. Należy ponadto ograniczyć korzystanie z kontrolek ActiveX na etapie rozruchu aplikacji. Aby zmniejszyć uciążliwość wolno uruchamiających się programów dla ich użytkownika, można w czasie ich rozruchu wyświetlać różnego rodzaju „ekrany rozbiegowe" (ang. *splash screens*);
- obciążenie sieci — dane powinny być wysyłane w większych porcjach, a otrzymywane odpowiedzi — blokowane;
- wykorzystanie baz danych — należy wykorzystywać tzw. aktualizacje buforowane (*cached updates*); czasochłonne zapytania powinny być realizowane w ramach oddzielnych wątków, zaś komunikaty o zaistniałych błędach — przetwarzane partiami. Dane pobierane w ramach pojedynczego zapytania powinny mieć rozsądnie ograniczony rozmiar, na przykład do wielkości pozwalającej na ich wyświetlenie w ramach jednego — dwóch ekranów. Ograniczanie widoczności danych powinno odbywać się raczej z użyciem zakresów (*ranges*) niż filtrowania (*filtering*).

Powyższe wskazówki mają oczywiście charakter ogólny; ich stosowalność do niektórych aplikacji może być mocno ograniczona, w niektórych przypadkach mogą one nie mieć w ogóle zastosowania.

Podsumowanie

W rozdziale tym zajęliśmy się różnymi aspektami optymalizowania aplikacji, zarówno pod kątem jej kompilowania i konsolidowania, jak też i szybkości oraz rozmiaru (i innych czynników) aplikacji końcowej. Wszelkie rozważania ilustrowane były na przykładzie średnio złożonej aplikacji, dla której zastosowanie opisanych zabiegów przyniosło skrócenie czasu wykonania o ponad trzy rzędy wielkości.

Powodzenie zabiegów optymalizacyjnych uwarunkowane jest dobrym zrozumieniem zarówno tego, co dzieje się w trakcie przekształcania kodu źródłowego na moduł wynikowy, jak i rozmaitych uwarunkowań i niuansów związanych z pracą nowoczesnych procesorów. W zakresie konkretnych problemów niezbędna jest ponadto wiedza o dostępności algorytmów, będących do dyspozycji przy ich rozwiązywaniu, jak również umiejętność efektywnej implementacji tych algorytmów w konkretnym języku programowania.

[8] Dokładne omówienie znaczenia adresu ładowania bazowego biblioteki DLL znajduje się na stronach 394 – 395 książki *Delphi 4. Vademecum profesjonalisty* (Helion, Gliwice 1999) — *przyp. tłum.*