Міністерство освіти і науки України
Чернівецький національний університет
імені Юрія Федьковича

Т.М. Сопронюк

Технології візуального й узагальненого програмування в C++Builder

Навчальний посібник

Чернівці
ЧНУ
2009

ББК 32.973.2 - 018я7
С 646
УДК 004.432 С++ (07)

Друкується за ухвалою редакційно-видавничої ради
Чернівецького національного університету
імені Юрія Федьковича

Сопронюк Т.М.

С646 Технології візуального й узагальненого програмування в C++Builder: Навчальний посібник. – Чернівці: ЧНУ, 2009. – 80 с.

У навчальному посібнику коротко описано сучасні інформаційні технології, які підтримує система програмування Borland C++Builder.

Детально розглянуто практичне застосування принципів узагальненого програмування на прикладі засобів стандартної бібліотеки шаблонів STL.

Візуальні технології програмування у середовищі C++Builder реалізуються засобами бібліотеки візуальних компонентів VCL (Visual Component Library), тому у посібнику широко висвітлюються можливості цієї бібліотеки для побудови проектів різного призначення, зокрема, графічних додатків і додатків баз даних.

Наведено по 30 варіантів завдань для кожної з чотирьох лабораторних робіт. Теми лабораторних робіт охоплюють описані у посібнику можливості системи програмування C++Builder.

Для студентів напряму підготовки "Прикладна математика" освітньо-кваліфікаційних рівнів "спеціаліст" і "магістр".

ББК 32.973.2 - 018я7
УДК 004.432 С++ (07)

Зміст

Сучасні технології програмування та C++Builder

Borland C++Builder – це одна з найпотужніших систем, що дозволяють на найсучаснішому рівні створювати як окремі прикладні програми Windows, так і розгалужені комплекси, призначені для роботи в корпоративних мережах і в Інтернет [1].

Серед нових можливостей C++Builder 6 можна відзначити кросплатформенні додатки, технології доступу до даних ADO, dbExpress, InterBaseExpress, компоненти-сервери COM, технології розподілених додатків COM, CORBA, MIDAS, нову методику диспетчеризації дій, програмування обміну інформацією по каналах зв'язку з використанням протоколів WebBroker, InternetExpress і XML, програмування Web-додатків.

Нова версія продукту C++Builder 2007 поєднує підтримку операційної системи Windows Vista, технологій Web 2.0, включаючи AJAX, із найновішими стандартами ANSI C++, включаючи нову бібліотеку підтримки Dinkumware. Продуктивність продукту зросла в п'ять разів у порівнянні з попередніми версіями. Розробникам запропоновані додаткові можливості візуалізації програмного коду й інтегровані засоби модульного тестування коду на C++. C++Builder поєднує у собі зручності візуального середовища розробки, об'єктно-орієнтований підхід, різноманітні можливості повторного використання коду, відкриту архітектуру й високопродуктивні компілятори мов Object Pascal і C++, які є одними із самих популярних мов програмування, а також масштабований доступ до даних, що зберігається в різних СКБД, як настільних, так і серверних.

Серед всіх сучасних технологій, які підтримує C++Builder, розглянемо коротко тільки ті технології програмування, які необхідно застосовувати при виконанні запропонованих у посібнику лабораторних робіт.

Об'єктно-орієнтоване програмування (ООП)

Об'єктно-орієнтоване програмування – це методика, що концентрує основну увагу програміста на зв'язках між об'єктами, а не на деталях їхньої реалізації.

У середовищі C++Builder класичні принципи ООП [11-13, 23, 30] (інкапсуляція, успадкування, поліморфізм, створення класів і об'єктів) інтерпретуються й доповнюються новими поняттями й термінологією, прийнятими інтегрованим середовищем візуальної розробки (IDE).

Мова C++ у середовищі C++Builder розширена новими можливостями (компоненти, властивості, методи, обробники подій) і останніми доповненнями стандарту ANSI C++ (шаблони, простори імен, явні й непостійні оголошення, ідентифікація типів при виконанні програми RTTI, виключення, динамічні масиви, ключові словами explicit, mutable, typename, automated і ін.). У C++Builder введено нові типи даних: булевий тип даних bool, рядковий тип AnsiString (реалізований як клас, оголошений у заголовочному файлі vcl/dstring.h), тип "множина" (реалізований як шаблон класу, оголошений у заголовочному файлі vcl/sysdefs.h) та інші типи.

Вважається [1, 2], що останні версії C++Builder найбільш повно відповідають стандарту ANSI/ISO серед всіх компіляторів на платформі Windows.

Компанія Borland пропонує потужний набір власних розширень мови C++, що забезпечує підтримку VCL бібліотеки (Visual Component Library). У C++Builder поряд із звичайними об'єктними класами мови C++, з'явилися нові компонентні класи (компоненти).

Форми є основою додатків C++Builder. Створення користувацького інтерфейсу додатка полягає в додаванні на форму елементів (компонентів). Компоненти C++Builder розташовуються на палітрі компонентів, виконаної у вигляді багатосторінкового блокнота. Важлива особливість C++Builder полягає в тому, що він дозволяє створювати власні компоненти й настроювати палітру компонентів, а також створювати різні версії палітри компонентів для різних проектів.

Компонент (component) – спеціальний клас, властивості якого подають атрибути об'єкту, а його методи реалізують операції над відповідними екземплярами компонентних класів. Поняття властивість, метод, обробник подій розглянемо далі, а зараз перелічимо основні відмінності компонентних класів від об'єктних класів [3, 27]:

- всі компоненти є прямими або непрямими нащадками одного загального класу-прародича (TComponent);
- компоненти звичайно використовуються безпосередньо, шляхом маніпуляції з їхніми властивостями;
- компоненти розміщаються тільки в динамічній пам'яті купи (heap) за допомогою оператора new, а не на стеку, як об'єкти звичайних класів;

- компоненти можна добавляти до палітри компонентів і далі маніпулювати з ними за допомогою редактора форм інтегрованого середовища візуальної розробки C++Builder;
- оголошення (інтерфейсна частина) компонентного класу обов'язково описується окремо (розширення .h) від реалізації (розширення .cpp).

Оголошення компонентного класу має вигляд:

```
class className : [специфікатор доступу:] parentClass
 {
    <Обява дружніх класів>
private:  //внутрішні деталі реалізації
    < приватні члени-дані,конструктори, методи>
protected:  //інтерфейс розробника
    <захищені члени-дані, конструктори, методи>
public:    // run-time інтерфейс
    <загальнодоступні властивості, члени-дані,
    конструктори, деструктор, методи>
__published:   //design-time інтерфейс
    <загальновідомі властивості, члени-дані>
    <Об'ява дружніх функцій>
};
```

Зазначимо, що parentClass – ім'я базового класу – обов'язково присутнє, оскільки будь-який компонент є прямим або непрямим нащадком класу TComponent.

Нова секція з ключовим словом __published – це доповнення, яке C++Builder вводить у стандарт ANSI C++ для оголошення загальновідомих елементів компонентних класів. Ця секція відрізняється від загальнодоступної тільки тем, що компілятор генерує інформацію про властивості, члени-дані та методи компонента для інспектора об'єктів.

Форма – це також компонент, і тому з кожною формою пов'язані файл оголошення та файл реалізації. Скелети обох файлів автоматично генеруються середовищем C++Builder при візуальному створенні нової форми, а програміст доповнює ці файли власним кодом. При додаванні у форму будь-якого компонента з палітри компонентів C++Builder автоматично формує програмний код для створення об'єкта (змінної) даного типу. Змінна додається як член класу даної форми.

Візуальне програмування інтерфейсів програмного забезпечення

Візуалізація – це процес графічного відображення складних процесів (у цьому випадку побудови) на екрані комп'ютера у вигляді графічних примітивів (графічних фігур) [24]. Візуалізовувати можна будь-які процеси: управління, побудови, малювання та інші. Приклад найпростішого варіанта візуалізації – лінійка прогресу (прямокутник, відсоток заповнення якого прямо пропорційний об'єму виконання якої-небудь операції). Дивлячись на лінійку, можна оцінити об'єм невиконаних операцій, який залишився.

Візуалізовувати можна інтерфейси програмного забезпечення. Це дозволяє спростити взаємодію програмного продукту з користувачем. Зображення на елементах інтерфейсу (зовнішнього вигляду програмного забезпечення) дозволяють користувачу інтуїтивно розбиратися в призначенні цих елементів.

Для візуалізації інтерфейсів програмного забезпечення існує цілий ряд спеціально розроблених елементів інтерфейсу – візуальних компонентів, що дозволяють відображати різну інформацію й здійснювати керування програмою в цілому. Найпростіший приклад – візуальна кнопка на екрані комп'ютера. Візуальна кнопка імітує поведінку звичайної кнопки на пульті керування будь-якого приладу. Її можна "натискати" як справжню.

Саме наявність візуальних засобів побудови інтерфейсів для Windows в C++Builder, а також створюване ними візуальне програмне забезпечення закріпили за ним термін "***візуальне програмування***".

Визначальними елементом процесу візуалізації є візуалізована модель, тобто модель, що піддається відображенню з метою можливості зміни її структури або її параметрів (або параметрів її окремих частин). Візуалізованою моделлю в C++Builder є вікно (форма, діалог) Windows, а не код програми. У C++Builder прийнято візуалізовувати тільки роботу з елементами інтерфейсу, коли об'єкти візуалізації розглядаються як візуальні компоненти, з яких складаються форми (вікна й діалоги) інтерфейсу програми.

Інструменти візуальної розробки в середовище C++Builder включають в себе дизайнер форм, інспектор об'єктів, палітру компонентів (вікно, що містить набір компонентів, з яких будується візуальна модель), менеджер проектів і редактор коду.

Ці інструменти включені в інтегроване середовище системи (Integrated Development Environment, IDE), яке забезпечує продуктивність багаторазового використання візуальних компонентів у поєднанні з удосконаленими інструментами й засобами доступу до баз даних.

Конструювання способом "перетягування" (drag-and-drop) дозволяє створювати додаток простим перетаскуванням захоплених мишею візуальних компонентів з палітри на форму додатка.

Механізми двунапрямної розробки (Two-Way-Tools) забезпечують контроль коду за допомогою гнучкої, інтегрованої й синхронізованої взаємодії між інструментами візуального проектування й редактором коду.

Інспектор об'єктів надає можливість оперувати властивостями (вікно властивостей – вікно, у якому відображаються параметри обраного елемента візуальної моделі) й подіями компонентів.

Компоненти можуть бути ***візуальні***, видимі при роботі додатку, і ***невізуальні***, виконуючі ті або інші службові функції. Візуальні компоненти відразу видні на екрані у процесі проектування у такому ж вигляді, в якому їх побачить користувач під час виконання додатку. Це дозволяє дуже легко вибрати місце їхнього розташування та їхній дизайн – форму, розмір, оформлення, текст, колір та інше. Невізуальні компоненти видимі на формі лише в процесі проектування у вигляді піктограм, але для користувача під час виконання вони не видимі, хоча й виконують для нього за кадром корисну роботу.

Візуальне програмування дозволяє звести проектування користувацького інтерфейсу до простих і наочних процедур. Але переваги візуального програмування не зводяться лише до цього.

Саме головне полягає в тому, що під час проектування форми й розміщення на ній компонентів C++Builder автоматично формує коди програми, включаючи в неї відповідні фрагменти, що описують даний компонент. А потім у відповідних діалогових вікнах користувач може змінити задані за замовчуванням значення властивостей цих компонентів і, при необхідності, написати обробники потрібних подій. Тобто проектування зводиться, фактично, до розміщення компонентів на формі, завдання деяких їхніх властивостей і написання, при необхідності, обробників подій. Результатом візуального проектування є скелет майбутньої програми, у яку вже внесені відповідні коди.

Технологія швидкої розробки додатків (RAD)

Завдяки візуальному об'єктно-орієнтованому програмуванню (ООП) була створена технологія, що одержала назву ***швидка розробка додатків*** *(RAD* — Rapid Application Development). Ця технологія характерна для нового покоління систем програмування, до якого відноситься й C++Builder [6, 7, 29].

C++Builder – це інструмент для швидкої розробки додатків (RAD) на C++ під Windows, який підтримує можливість програмування, що ґрунтується на компонентах.

Компоненти – це будівельні блоки для додатків. Тобто, використовуються об'єкти-компоненти зі своїми можливостями і об'єднуються в один додаток. C++Builder сам побудований на компонентах і робота з компонентами в C++Builder проста і надійна.

Швидка розробка додатків (RAD) означає підтримку властивостей, методів і подій компонентів у рамках об'єктно-орієнтовного програмування.

Властивості дозволяють легко встановлювати різноманітні характеристики компонентів, такі як назви, контекстні підказки або джерела даних. Методи (функції-члени) роблять певні операції над компонентним об'єктом. Події зв'язують впливи користувача на компоненти із кодами реакції на ці впливи.

Працюючи спільно, властивості, методи і події утворять середовище RAD інтуїтивного програмування надійних додатків для Windows [3].

Як було сказано вище, компонент – це спеціальний клас, властивості якого подають атрибути об'єкту, а його методи реалізують операції над відповідними екземплярами компонентних класів.

Якщо вибрати компонент із палітри й додати його до форми, інспектор об'єктів автоматично покаже властивості й події, які можуть бути використані із цим компонентом. У верхній частині інспектора об'єктів є список, що випадає, що дозволяє вибирати потрібний об'єкт із наявних на формі.

Поняття ***метод*** звичайно використовується в контексті компонентних класів і зовнішньо не відрізняється від терміна ***функція-член*** звичайного класу. Щоб викликати метод, треба вказати ім'я функції в контексті даного класу. Саме схований зв'язок методу з класом виділяє його з поняття простої функції. Під час виконання методу він має доступ до всіх даних свого класу. Це забезпечується

передачею кожному методу схованого параметра – вказівника **this** на екземпляр класу. При будь-якому звертанні методу до членів даних класу, компілятор генерує спеціальний код, що використовує вказівник **this.** Метод є функцією, що пов'язана з компонентом і оголошується як частина об'єкта. Методи можна викликати, використовуючи операцію доступу через вказівник -> для цього компонента.

C++Builder дозволяє маніпулювати виглядом і функціональною поведінкою компонентів не тільки за допомогою методів (як це роблять функції-члени звичайних класів), але й за допомогою властивостей і подій, властивих тільки класам компонентів. Маніпулювати з компонентним об'єктом можна як на стадії проектування додатка, так і під час його виконання.

Властивості (properties) компонентів являють собою розширення поняття членів даних і, хоча не бережуть дані як такі, проте забезпечують доступ до членів даних об'єкта. C++Builder використовує ключове слово **__property** для оголошення властивостей. Властивості є атрибутами компонента [8], що визначають його зовнішній вигляд і поведінку. Багато властивостей компонента мають значення, за замовчуванням (наприклад, висота кнопок). Властивості компонента відображаються на сторінці властивостей (Properties). Інспектор об'єктів відображає опубліковані (published) властивості компонентів. Крім published-властивостей, компоненти можуть і найчастіше мають загальні (public) властивості, які доступні тільки під час виконання додатка. Інспектор об'єктів використовується для встановлення властивостей під час проектування. Список властивостей розташовується на сторінці властивостей інспектора об'єктів. Можна визначити властивості під час проектування або написати код для видозміни властивостей компонента під час виконання додатка.

При визначенні властивостей компонента під час проектування потрібно вибрати компонент на формі, відкрити сторінку властивостей в інспекторі об'єктів, вибрати потрібну властивість і змінити її за допомогою редактора властивостей (це може бути просте поле для введення тексту або числа; список, що випадає; список, що розкривається діалогова панель та інше).

Сторінка подій (**Events**) інспектора об'єктів показує список подій, розпізнаваних компонентом. За допомогою ***подій (events)*** компонент повідомляє користувачу про те, що на нього зроблений деякий визначений вплив (програмування для операційних систем із

графічним користувацьким інтерфейсом, зокрема, для Windows XP або Windows NT передбачає опис реакції додатка на ті або інші події, а сама операційна система займається постійним опитуванням комп'ютера з метою виявлення настання якої-небудь події). Кожний компонент має свій власний набір обробників подій. У C++Builder необхідно писати функції, які називаються обробниками подій, і зв'язувати події із цими функціями. Якщо подія відбудеться і обробник цієї події написаний, то програма виконає написану функцію.

Для того, щоб додати обробник подій, потрібно вибрати на формі за допомогою миші компонент, якому необхідний обробник подій, потім відкрити сторінку подій інспектора об'єктів і двічі клацнути лівою клавішею миші на колонку значень поруч із подією, щоб C++Builder згенерував прототип обробника події і показав його в редакторі коду. При цьому автоматично генерується текст порожньої функції, і редактор відкривається в тому місці, де треба вводити код. Курсор позиціонується усередині операторних дужок { ... }. Далі потрібно ввести код, який повинен виконуватися при настанні події. Обробник подій може мати параметри, які вказуються після імені функції в круглих дужках.

C++Builder використовує ключове слово **__closure** для оголошення подій. Основна сфера застосування методів – це ***обробники подій (event handlers)***, що реалізують реакцію програми на виникнення визначених подій. Типові прості події – натискання кнопки або клавіші на клавіатурі.

Узагальнене програмування (generic programming)

Поряд з ООП у C++Buider широко підтримується ***узагальнене програмування*** [15], яке спрямоване на спрощення повторного використання коду і на абстрагування загальних концепцій. Якщо в ООП акцент програмування ставиться на дані, то в узагальненому програмування – на алгоритми.

Узагальнене програмування має справу з абстрагуванням і класифікацією алгоритмів і структур даних. У багатьох випадках алгоритм можна виразити незалежно від деталей подання оброблюваних ним даних. Термін ***узагальнений*** приписується коду, тип якого є незалежним. В узагальненому програмуванні можна один раз написати функцію для узагальненого, тобто невизначеного типа і потім використовувати її для множини існуючих типів.

Концепція узагальненого програмування припускає використання типів даних як параметрів. При розробці алгоритму, що може працювати із множиною типів і структур даних, використовується якийсь абстрактний тип, що згодом параметризується. Такий підхід забезпечує простий спосіб введення різного роду загальних концепцій і позбавляє програміста від написання вручну спеціалізованого коду.

У мові програмування C++ узагальнене програмування реалізується за допомогою шаблонів. Шаблон являє собою параметризоване визначення деякого елемента програми. При цьому розроблювач усього лише вказує компілятору правило для генерації (породження) одного або декількох конкретних екземплярів цього елемента програми. У C++ допускається створення шаблонів для функцій, методів і класів. При цьому як параметр шаблона можна використовувати тип (клас), звичайний параметр відомого типу, інший шаблон. У шаблона може бути кілька параметрів.

Поряд із засобами визначення власних користувацьких шаблонів у розпорядження розробника надається стандартна бібліотека. У ній визначені шаблони, що представляють найбільш використовувані структури даних, а також алгоритми для їхньої обробки.

Отже, найбільш перспективним стає наступний (більш високий у порівнянні із класами) рівень абстрактного програмування – створення своїх і використання стандартних шаблонів і узагальнених алгоритмів стандартної бібліотеки, які вже розроблені й налагоджені професіоналами. Тому далі розглянемо стандартну бібліотеку шаблонів (STL).

Використання стандартної бібліотеки шаблонів STL

Огляд стандартної бібліотеки шаблонів

Перелічимо основні можливості стандартної бібліотеки C++, які з'явилися в ній з розвитком самої мови C++: потоки, числові методи (наприклад, операції з комплексними числами, множення масиву на константу та інші нескладні методи), рядки (класи для роботи з текстовою інформацією), множини, контейнери, алгоритми й об'єкти-функції, ітератори та інше.

Стандартна бібліотека шаблонів STL [14-21, 30] – частина стандартної бібліотеки C++ SCL, що забезпечує загальноцільові

стандартні класи й функції, які реалізують найбільш популярні й широко використовувані алгоритми й структури даних.

Бібліотека STL розроблена співробітником Hewllet-Packard Олександром Степановим. STL будується на основі шаблонів класів, і тому вхідні в неї алгоритми й структури застосовні майже до всіх типів даних. Ядро бібліотеки утворюють три складові: контейнери, алгоритми й ітератори.

Контейнери (containers) – це об'єкти, призначені для зберігання інших елементів. Тут представлені різні варіанти часто використовуваних структур даних – вектори, списки, стеки, черги й т.п. Точніше, навіть не самі структури, а заготівки для них. Конкретну структуру для свого типу даних програміст утворює сам за допомогою стандартних контейнерів.

Ітератори (iterators) – це об'єкти, які стосовно контейнера відіграють роль вказівників. Вони дозволяють одержати доступ до вмісту контейнера приблизно так само, як вказівники використовуються для доступу до елементів масиву. Ітератори застосуються як зв’язок між структурами даних (не обов'язково контейнерами) і алгоритмами або іншим кодом, який ці структури використовує.

Алгоритми (algorithms) виконують операції над містимим контейнера. Існують алгоритми для ініціалізації, сортування, пошуку, заміни вмісту контейнерів. Багато алгоритмів призначені для роботи з послідовністю (sequence), що являє собою лінійний список елементів усередині контейнера. Велика частина алгоритмів STL побудована по єдиному принципу. Алгоритм отримує на вхід пару ітераторів (інтервал) і для елементів з цього інтервалу виконує деяку задачу, наприклад, сортування.

Шаблони контейнерів, алгоритми та й взагалі вся бібліотека винесені в окремий простір імен, який одержав назву std. Існує кілька способів користуватися ключовими словами із простору імен std:

- після всіх файлів, що підключаються (наприклад, #include<set>), використати директиву

```
using namespace std;
```

- підключити бажані модулі STL окремими директивами using:

```
using std::stack;
using std::find;
```

- вказувати оператор прив'язки std:: перед кожним STL типом даних або алгоритмом.

Контейнери

Контейнер – це сховище елементів і засобів, щоб із цими елементами працювати. Всі контейнери розроблені таким чином, щоб при необхідності можна було працювати з елементами незалежно від того, у якому саме різновиді контейнера – векторі, списку, чи черзі – ці елементи зберігаються. Природно, різновиди розрізняються між собою й внутрішнім устроєм, і ефективністю різних операцій, і набором додаткових, характерних тільки для них способів доступу до елементів. Наприклад, у вектора є оператор індексування [], якого немає у списків, проте список дозволяє ефективно виконувати операції вставки й видалення елементів. Але при цьому кожній з контейнерів підтримує набір однотипних операцій – невеликий, але достатній для того, щоб на його основі можна було писати узагальнені алгоритми для роботи з елементами.

У мові С був лише один контейнер – масив. У мові С++ контейнери реалізовані у вигляді шаблонів класів, при цьому тип елемента задається параметром шаблона.

Стандартна бібліотека шаблонів містить наступні контейнерні класи: vector (динамічний масив), string (рядок), list (список), deque (дек), set (множина), multiset (мультимножина), bitset (множина бітів), map (асоціативний масив), multimap (мульти-асоціативний масив). Крім того, класи queue (черга), priority_queue (черга з пріоритетом) і stack (стек) не є окремими контейнерами, а побудовані на базі інших контейнерів. Для використання контейнерів необхідно підключити однойменний заголовний файл (наприклад, для використання списків варто використовувати #include <list>).

Розглянемо далі основні контейнери, структури і класи стандартної бібліотеки.

vector<T> – контейнерний шаблон, що описує динамічний одновимірний масив елементів типу T. Визначений у файлі заголовків <vector>. Відмінна риса від інших типів контейнерів – наявність ефективної операції доступу по індексу (operator[]). Цей контейнер не є зручним при вставці й видаленні елементів.

Наведемо приклад роботи з вектором і матрицею з використанням бібліотеки STL.

```
#include <vector>
#include <iostream>
// використовується простір імен бібліотеки STL
using namespace std;
    .  .  .
// створюється вектор з 10 елементів типу int
vector<int> v(10);
  for (int i=0; i<10; i++)
    v[i] = i;  // використовується operator[]
// створюється матриця 10x10 елементів типу int
 vector<int> m[10];
 for(int i=0;i < 10;i++)
    for(int j=0;j < 10;j++)
    // використовується член-функція push_back
    m[i].push_back(j);
for(int i=0;i < 10;i++)
{
for(int j=0;j < 10;j++) cout<<m[i][j]<<" ";
cout<<endl;
}
    .  .  .
```

Матрицю можна описати і так:

```
vector< vector<int> > m;
```

Оскільки дві кутові дужки, що йдуть підряд без пробілу, більшість трансляторів сприймають як операцію «<<» або «>>», то, при наявності в коді вкладених STL конструкцій, між кутовими дужками треба ставити пробіл.

Для спрощення описів можна визначити типи, наприклад так:

```
typedef vector<int> tvector;       // тип вектор
typedef vector<tvector> tmatrix;  // тип матриця
```

string. Для роботи з рядками передбачений шаблонний клас basic_string з якого визначається клас string:

```
typedef basic_string <char> string;
```

Клас string відрізняється від vector<char> функціями для маніпулювання рядками й політикою роботи з пам'яттю. Для визначення довжини рядка, використовується string::length(), а не vector::size().

Розглянемо приклад роботи з рядками.

```
string s1 = "слово";
string s1 = s1.substr(0, 3),    // "сло"
       s2 = s1.substr(1, 3),    // "лов"
       s3 = s1.substr(0, s.length()-1),    //"слов"
       s4 = s1.substr(1),    // "лово"
       s = "Завтра - термін здачі лабораторної !";
// пошук першого пробілу
int ind1 = s.find_first_of(' ');
// пошук першого пробілу, знаків ! або -
int ind2 = s.find_first_of(" -!");
```

Також рядки можна додавати за допомогою operator+, порівнювати за допомогою операторів порівняння. Таким чином, vector<string> можна впорядковувати стандартними методами. Рядки порівнюються в лексикографічному порядку.

pair<T1,T2> – пара. Шаблонна структура, що містить два поля, можливо, різних типів. Поля мають назви first і second. Прототип пари виглядає так:

```
template<class T1, class T2> struct pair {
   T1 first;
   T2 second;
    pair() {}
    pair(const T1& x, const T2& y) :
                        first(x), second(y) {}
};
```

Прикладом контейнера пара є опис pair<int,int> – два цілих числа. Приклад складеної пари: pair<string, pair<int,int> > – рядок і два цілих числа. Використовувати подібну пару можна, наприклад, так:

```
pair<string, pair<int,int> > P;
string s = P.first; // Рядок
int x = P.second.first; // Перше ціле
int y = P.second.second; // Друге ціле
```

Об'єкти pair можна порівнювати по полях у порядку опису пар ліворуч праворуч. Тому пари активно використовуються як усередині бібліотеки STL, так і програмістами у своїх цілях.

Наприклад, якщо необхідно впорядкувати точки на площині по полярному куту, то можна помістити всі точки в структуру виду

vector< pair<double, pair<int,int> >,

де double – полярний кут точки, а pair<int,int> – її координати. Після цього викликати стандартну функцію сортування, і точки будуть упорядковані по полярному куту.

Пари активно використовуються в асоціативних масивах.

list<T> – контейнерний шаблон, що описує список елементів типу T. Визначається у файлі заголовків <list>. Зручний при частих вставках і видаленнях елементів.

deque<T> – дек (черга із двома кінцями і елементами типу T). Робота з елементами на обох кінцях (але не в середині) майже так само ефективна, як у списку, а доступ по індексу [] – як у векторі. Слабке місце – вставка й видалення елемента всередині.

stack<T>, **queue<T>** – стек і черга з елементів типу T. Визначаються у файлах заголовків відповідно <stack> і <queue>. Контейнер, на базі якого будуються стек і черга, є параметром шаблона. За замовчуванням стек і черга використовують для реалізації дек (тоді в кутових дужках задається лише один параметр – тип елементів T, другий параметр – за замовченням deque<T>). Однак, це можна поміняти, указавши другим (необов'язковим) параметром тип іншого контейнера. Для стека це може бути список або вектор (обидва підтримують push_back), а для черги – список (підтримує pop_front):

```
stack<int,vector<int> > s;
queue<double,list<double> > q;
```

set<T>– контейнерний шаблон, що описує множину елементів типу T. Визначаються у файлі заголовку <set> .

map <T1,T2> – контейнерний шаблон, що описує асоціативний масив. Асоціативний масив – це масив, у якого як індекс типу T1 (ключ) може використовуватися значення нецілочислового типу,

наприклад, рядок. T2 – це тип самих елементів. Асоціативний масив багато в чому схожий на звичайну послідовність, але дозволяє працювати з парами джерело-значення. Втім, у деяких асоціативних контейнерах значень, як таких, немає – є тільки ключі.

Конструктори та члени-функції контейнерів

Для створення будь-якого з контейнерів може бути використаний конструктор за замовчуванням (у цьому випадку створюється порожній контейнер), конструктор копіювання, конструктор, який ініціалізує контейнер діапазоном елементів іншого контейнера (не обов'язково того ж типу):

```
container<T> c;
container<T> c2(c);
container<T> c1(b,e);
```

Тут container – будь-який з типів контейнерів STL, а c, c1 і c2 – утворені контейнери.

Загальні функції для всіх контейнерів

c.begin()	повертає ітератор початку контейнера
c.end()	повертає ітератор кінця контейнера
c.rbegin()	повертає зворотний ітератор початку контейнера
c.rend()	повертає зворотний ітератор кінця контейнера
c.size()	повертає розмір контейнера
c.empty()	перевіряє, чи порожній контейнер
c.clear()	очищає контейнер, роблячи його розмір рівним 0
c=c2	присвоює вміст контейнера c2 однотипному контейнеру c. Використовується поелементне присвоювання
c.assign(b,e)	заміняє вміст контейнера елементами діапазону [b,e) іншого контейнера
c.swap(c1)	міняє місцями вміст двох однотипних контейнерів (відбувається обмін внутрішніми вказівниками)
c==c1 c!=c1	порівнює контейнери на рівність і нерівність, використовуючи операції == і != для елементів
c<c1 c>c1 c<=c1 c>=c1	порівнює контейнери лексикографічно

Функції для контейнерів vector і list

container<T> c(n)	конструктор контейнера з n елементів зі значенням за замовчуванням
container<T> c(n,t)	конструктор контейнер з n елементів, заповнених значенням t
c.insert(it,t)	вставляє елемент t у контейнер c перед ітератором it. Повертає ітератор вставленого елемента.
c.insert(it,n,t)	вставляє n елементів t у контейнер c перед ітератором it. Повертає ітератор першого вставленого елемента.
c.insert(it,b,e)	вставляє діапазон елементів [b,e) у контейнер c перед ітератором it. Повертає ітератор першого вставленого елемента.
c.erase(it)	видаляє елемент у контейнері c в позиції ітератора it. Повертають ітератор елемента, що слідує за вилученим
c.erase(b,e)	видаляє діапазон елементів [b,e) у контейнері c Повертають ітератор елемента, що слідує за вилученими
c.assign(n,t)	присвоює контейнеру c n копій t
c.front()	повертає посилання на перший елемент
c.back()	повертає посилання на останній елемент
c.push_back(t)	додає t у кінець контейнера c
c.pop_back()	видаляє останній елемент. Повертає void

Функції для контейнерів vector, string і deque

c[i]	здійснює довільний доступ до і-того елемента без перевірки виходу за межі
c.at(i)	здійснює довільний доступ з перевіркою виходу за межі (у випадку виходу генерується виключення out_of_range)

Функції для контейнерів vector і string

c.resize(n)	змінює розмір
c.capacity()	повертає пам'ять, відведену під масив
c.reserve(n)	резервує пам'ять в n елементів

Функції для контейнерів list

c.sort()	сортування списку c
c.sort(Cmp)	сортування списку c із критерієм сортування Cmp (аналог операції <)
c.push_front(t)	додавання елемента t у початок
c.pop_front()	видалення першого елемента
c.splice(it,c1)	переміщення всіх елементів зі списку c1 у список c перед ітератором it (список c1 обнулюється)
c.splice(it,c1,it1)	переміщення елемента *it1 зі списку c1 у список c перед ітератором it
c.splice(it,c1,b,e)	переміщення елементів [b,e) зі списку c1 у список c перед it
c.merge(c2)	об'єднання двох сортованих списків з видаленням елементів з c2 і вставкою в c с збереженням порядку
c.remove(t)	видалення всіх елементів t
c.remove_if(Pred)	видалення всіх елементів, що задовольняють предикату Pred
c.unique()	видалення рядом розташованих дублікатів, використовуючи ==
c.unique(BinPred)	видалення рядом розташованих дублікатів, використовуючи бінарний предикат BinPred як функцію порівняння на рівність

Ітератори

Ітератори – це щось, аналогічне вказівнику на елемент масиву. По суті, вказівник на елемент масиву – теж ітератор. Маючи ітератор p якого-небудь контейнера, можна перейти до наступного або попереднього елемента (++p, --ip), одержати сам елемент (*p), порівняти з іншим ітератором того ж контейнера (p==p1). Але ітератори відрізняються від звичайних вказівників звуженими можливостями. Наприклад, не з всіма типами ітераторів можна робити операцію декремента. Деякі ітератори дозволяють працювати з об'єктами в режимі "тільки читання" або "тільки запис".

Отже, ітератори потрібні, щоб давати зручний і однаковий доступ до елементів будь-якого контейнера. Їхнє головне призначення – задавати послідовності елементів, тобто – ті границі (по елементах), у межах яких треба працювати. Для звичайного масиву можна обійтися не ітераторами, а індексами. Але, якщо треба

вибрати з асоціативного масиву із ключем-рядком всі елементи з індексом, що починається на А, то треба використати ітератор.

Існує багато різновидів ітераторів – прямі, зворотні, двунапрямні, з довільним (випадковим) доступом та інші [19].

Тип ітератора	***Доступ***	***Розіменування***	***Ітерація***	***Порівняння***
Ітератор виводу (output iterator)	Тільки запис	*	++	
Ітератор введення (input iterator)	Тільки читання	*, ->	++	==, !=
Прямий ітератор (Forward iterator)	Читання й запис	*, ->	++	==, !=
Двунапрямний ітератор (bidirectional iterator)	Читання й запис	*, ->	++, --	==, !=
Ітератор з довільним доступом (random-access iterator)	Читання й запис	*, ->, []	++, ---,+, -, +=, -=	==, !=, <, <=, >, >=

У різних алгоритмах використовуються різні типи ітераторів.

Ітератор зовсім необов'язково використовувати з повною версією стандартного контейнера. Деякі алгоритми працюють і зі звичайними масивами, при цьому в якості ітераторів їм передаються вказівники на елементи масиву.

Ітератори контейнерів реалізовані у вигляді шаблонів, причому вони є частиною визначення класу самого контейнера. Описати ітератор можна, наприклад, так:

```
list<int>::iterator p;
vector<int>::revers_iterator r;
```

Тепер можна звертатися до елементів списка через ітератор

```
list<int> res = v;
for (p=res.begin(); p!=res.end(); p++)
     cout << *p << endl;
```

Тут v – будь-який контейнер, наприклад, вектор.

Алгоритми

Алгоритми (і об'єкти-функції) потрібні для того, щоб можна було ефективно працювати з набором елементів контейнера. Але, не тільки контейнера. Завдяки ітераторам алгоритми є досить незалежними, їх можна відокремити від поняття "контейнер". Алгоритми просто обробляють елементи послідовності, заданої ітераторами.

Що саме треба проробити з послідовністю, задається самим алгоритмом. Поведінкою багатьох алгоритмів можна керувати, передаючи їм як аргументи функції (це можуть бути звичайні функції або функції-об'єкти).

З використанням алгоритмів можливе створення дуже могутніх і ефективних програм. По компактності такий код перевершує код, написаний на таких сучасних мовах, як Java і C#, і в значній мірі ефективніше останнього.

STL-алгоритми визначені в заголовному файлі <algorithm>. Вони представляють набір готових функцій, які можуть бути застосовані до STL-контейнерів (і не тільки) і можуть бути розділені на групи. Перелічимо основні алгоритми в групах.

Алгоритми, що не модифікують послідовностей елементів

for_earch ()	виконує операції для кожного елемента послідовності
find ()	знаходить перше входження значення в послідовність
find_if ()	знаходить першу відповідність предикату в послідовності
count ()	підраховує кількість входжень значення в послідовність
count_if ()	підраховує кількість виконань предиката в послідовності
search ()	знаходить перше входження послідовності як підпослідовності
search_n ()	знаходить n-те входження значення в послідовність

Алгоритми, що модифікують послідовностей елементів

copy()	копіює послідовність, починаючи з першого елемента
swap()	міняє місцями два елементи
replace()	заміняє елементи із зазначеним значенням
replace_if()	заміняє елементи при виконанні предиката
replace_copy()	копіює послідовність, замінюючи елементи з зазначеним значенням
replace_copy_if()	копіює послідовність, замінюючи елементи при виконанні предиката
fill()	заміняє всі елементи даним значенням
remove()	видаляє елементи з даним значенням
remove_if()	видаляє елементи при виконанні предиката
remove_copy()	копіює послідовність, видаляючи елементи з зазначеним значенням
remove_copy_if()	копіює послідовність, видаляючи елементи при виконанні предиката
reverse()	міняє порядок проходження елементів на зворотний
random_shuffle()	переміщає елементи відповідно до випадкового рівномірному розподілу (“тасує” послідовність)
transform()	виконує задану операцію над кожним елементом послідовності
unique()	видаляє рівні сусідні елементи
unique_copy()	копіює послідовність, видаляючи рівні сусідні елементи

Алгоритми сортування членів послідовностей

sort ()	сортує послідовність із задовільною середньої ефективністю
partial_sort ()	сортує частину послідовності
stable_sort ()	сортує послідовність, зберігаючи порядок слідування рівних елементів
lower_bound ()	знаходить перше входження значення у відсортованій послідовності
upper_bound ()	знаходить перший елемент, більший ніж задане значення
binary_search ()	визначає, є чи даний елемент у відсортованій послідовності

merge ()	зливає дві відсортовані послідовності

Алгоритми роботи з множинами

includes ()	перевірка на входження
set_union ()	об'єднання множин
set_intersection ()	перетин множин
set_difference ()	різниця множин

Мінімуми й максимуми

min ()	менше із двох
max ()	більше із двох
min_element ()	найменше значення в послідовності
max_element ()	найбільше значення в послідовності

Перестановки

next_permutation ()	наступна перестановка в лексикографічному порядку
pred_permutation ()	попередня перестановка в лексикографічному порядку

Для того, щоб використовувати ці та інші алгоритми треба мати відповідну документацію. У середовище C++Builder 6 включено документацію по бібліотеці STL. Необхідну інформацію можна також знайти на сайтах [16-21].

Для багатьох алгоритмів STL необхідно задати умову, за допомогою якої алгоритм визначає, що йому необхідно робити з тим або іншим членом контейнера. За означенням, предикат – це функція, що приймає один або більше параметрів і повертає значення істина або неправда. Існує набір стандартних предикатів.

Приклад використання контейнерів, ітераторів, алгоритмів і предикатів

Продемонструємо використання стандартної бібліотеки шаблонів STL на приклалі контейнера vector. Для цього створимо декілька об’єктів різними типами конструкторів, застосуємо до них алгоритми for_each, find, find_if (з користувацьким предикатом), find_end, reverse і функцію swap.

```
#include <iostream>
#include <vector> // для контейнера vector

// для алгоритмів for_each, find, find_if, find_end
#include <algorithm>

// використовується простір імен бібліотеки STL
using namespace std;

// функції, які застосовуються в алгоритмах
 void mul2(int& i) { i *= 2; }
 void print(int i) { cout << i << endl; }

// предикат для алгоритму find_if
bool negative(int x) { return x<0; }

int main(int argc, char* argv[]) {
  // створюється вектор з 10 елементів
  vector<int> v(10);

  // використовується operator[]
 for (int i=0; i<10; i++) v[i] = i;
  v[6]=-4;

  // створюється контейнер - копію першого
  // (конструктор копіювання)
  vector<int> res = v;

  // використовується ітератор для множення
  // всіх елементів res на 2 і друку масиву
  vector<int>::iterator p;
  for (p=res.begin(); p!=res.end(); p++)
   {
    *p *= 2;
   cout << *p << endl;
   }

// аналогічні дії робимо за допомогою
// стандартного алгоритму for_each
   for_each(res.begin(), res.end(), mul2);
   for_each(res.begin(), res.end(), print);

// стандартний алгоритм  find (шукає у масиві число 3)
    p = find(v.begin(), v.end(), 3);
```

```
  if (p != v.end())cout << "\nЗнайдено " << *p << endl;

 // пошук першого від'ємного елемента
 // (використовується предикат negative)
 p= find_if(v.begin(), v.end(), negative);
 if (p != v.end())
   cout << " Перший від\'ємний  елемент " << *p << endl;
 else  cout << " Від\'ємних елементів не знайдено \n" ;

  int subv[]={24,28,32,36};

 // ще один тип конструктора
   vector<int> sub(subv,subv+4);

 // пошук в res останньої послідовності sub
p=find_end(res.begin(),res.end(),sub.begin(),sub.end());

 if(p!= res.end())cout << "Послідовність знайдено \n";
      else  cout << " Послідовність не знайдено \n" ;

 res.swap (sub); //перестановка об\'єктів res і sub
 reverse (res);  // перестановка елементів вектора res

 cout << " Перетворений вектор res: " << endl;
 for_each(res.begin(), res.end(), print);

 //затримка екрану з результатами
 char c;    cin>>c;

 return 0;
 }
```

Використання бібліотеки візуальних компонентів VCL

Огляд бібліотеки VCL

Бібліотека візуальних компонентів VCL (Visual Component Library) – об'єктно-орієнтована бібліотека для розробки програмного забезпечення, розроблена компанією Borland для підтримки візуального програмування. VCL [6-10, 25-29] входить у комплект поставки C++Builder, Delphi і Borland Developer Studio і є частиною середовища розробки, хоча розробка додатків у цих середовищах можлива й без використання VCL. VCL представляє величезну кількість (більше 360) готових до використання компонентів для

роботи в самих різних областях програмування, таких, наприклад, як інтерфейс користувача (екранні форми, елементи керування та інші), робота з базами даних, взаємодія з операційною системою, програмування мережних додатків та інше.

Сукупність функцій, за допомогою яких здійснюється доступ до системних ресурсів, називається прикладним програмним інтерфейсом, або API (Application Programming Interface). Для взаємодії з Windows додаток використовує функції API, за допомогою яких реалізуються всі необхідні системні дії, такі як виділення пам'яті, вивід на екран, створення вікон і т.п. Бібліотека VCL є просто оболонкою для WinAPI. Класи VCL створені тільки для того, щоб полегшувати процес програмування. Наприклад, один з найважливіших класів TForm з бібліотеки VCL можна було б створити, не використовуючи класи VCL.

Компонент C++Builder – це особливий вид об'єктів – візуальний об'єкт (візуальний для проектування, а не для відображення користувача). Створювати й редагувати такий об'єкт можна як програмним шляхом, так і на етапі проектування.

При виконанні програми компоненти діляться на візуальні, які бачить користувач, і невізуальні, для яких немає можливості їхнього відображення, але доступ до властивостей яких дозволений.

Усі компоненти мають загального предка – клас TComponent. Усі компоненти C++Builder можуть бути доступні через палітру компонентів. Частина компонентів є елементами керування. В основному – це елементи керування Windows.

Елементи керування можна підрозділити на віконні й невіконні. Віконні елементи можуть одержувати фокус вводу і мають дескриптор вікна. Предком всіх віконних елементів керування є абстрактний клас TWinControl. Предком невіконних елементів керування є абстрактний клас TGraphicControl.

При додаванні у форму будь-якого компонента з палітри компонентів C++Builder автоматично формує програмний код для створення об'єкта (змінної) даного типу. Змінна додається як член класу даної форми.

Класи бібліотеки VCL використовують механізм простого успадкування: один клас може мати тільки одного предка. Коренем ієрархії класів є клас **TObject**. Нижче наведено ієрархію ключових базових класів бібліотеки VCL.

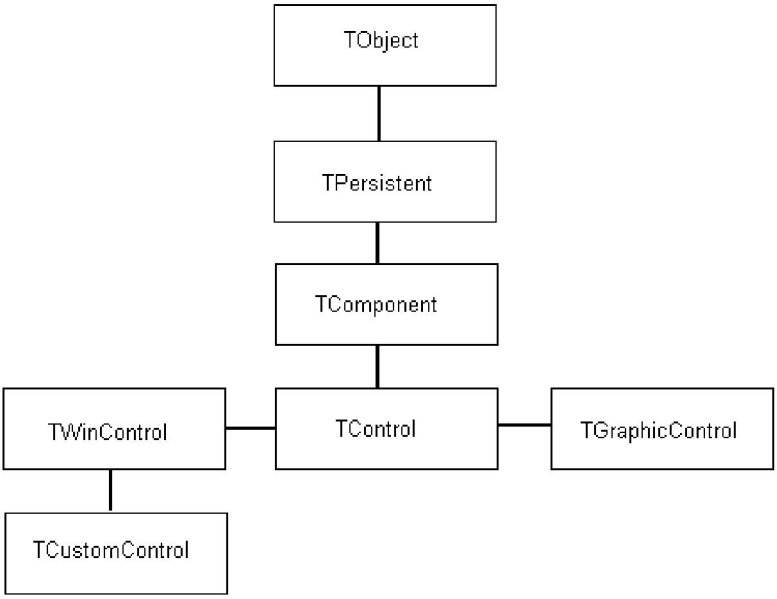

Рис. 1.

Будь-який ***клас*** VCL-бібліотеки успадковується від класу TObject. Клас **TPersistent** пішов безпосередньо від класу TObject. Він забезпечує своїх нащадків можливістю взаємодіяти з іншими об'єктами і процесами на рівні даних. Його методи дозволяють передавати дані в потоки, а також забезпечують взаємодію об’єкта з інспектором об'єктів. Клас TPersistent має метод Assign, який дозволяє передавати поля і властивості одного об’єкту іншому.

Клас **TComponent** є предком всіх ***компонентів*** VCL-бібліотеки. Всі нащадки даного класу можуть бути розташовані в палітрі компонентів.

Клас TComponent дозволяє визначати батьківський елемент керування й власника компонента. Батьківським елементом керування називається той, у який безпосередньо поміщений даний компонент. Власником всіх компонентів, розташованих у формі, є сама форма. Власником всіх форм є додаток.

Якщо компонент розташований не безпосередньо у формі, а, наприклад, у компоненті типу TPanel, то власник і батьківський елемент керування в нього будуть різні.

TControl – це базовий клас всіх елементів керування (включаючи й вікно форми). Ці компоненти можуть бути видимі під час виконання. Для них визначені такі властивості, як позиція, курсор, підказка, що спливає, методи для малювання або переміщення елемента керування, події для маніпуляцій за допомогою миші.

Клас **TWinControl** є базовим класом всіх віконних елементів керування.

Клас **TApplication** інкапсулює об'єкт "windows-додаток". За допомогою цього класу визначається інтерфейс між розробником і середовищем Windows. У кожному додатку C++Builder завжди автоматично створюється один об'єкт Application як екземпляр класу додатка. Для більшості додатків цей об'єкт є екземпляром класу TApplication. Компонент TApplication не відображається в палітрі компонентів і не має властивостей, що публікуються. Для того, щоб мати можливість перехоплювати події для додатка, можна додати в будь-яку форму проекту компонент TApplicationEvents.

Кожний додаток C++Builder має глобальну змінну Screen типу TScreen. Компонент **TScreen**, так само як і компонент TApplication, недоступний з інспектора об'єктів. Цей компонент призначений для забезпечення доступу до пристрою виводу – екрану. Його властивості містять інформацію про використовуване розширення монітора, курсорах і шрифтах, доступних для додатка, списку форм додатка й активній формі.

TForm є базовим класом для створення вікна форми. За замовчуванням кожна нова створювана форма реалізується як нащадок класу TForm. Форма може бути головним вікном додатка, діалоговим вікном, дочірнім або MDI-вікном.

Клас форми є контейнером для всіх компонентів, розташовуваних на формі. Для доступу до властивостей форми або іменам компонентів можна використовувати вказівник this. Якщо перед іменем властивості відсутній який-небудь вказівник, то за замовчуванням вважається, що це властивість форми.

Динамічні компоненти форми

Компоненти на форму можна добавляти не тільки під час проектування. Їх можна створювати і знищувати динамічно під час роботи програми.

Для того, щоб створити на формі новий екземпляр, наприклад, компонента типу TButton, спочатку в заголовочному файлі в секції private форми треба описати вказівник на майбутній компонент:

```
private:
    TButton *pButton;
```

Далі у файлі реалізації, можливо – у деякому обробнику подій, динамічно створювати сам компонент:

```
    // власник компонента - форма
    pButton = new pButton(this);

    //батько компонента – форма (задається обов'язково)
    pButton->Parent = this;

    // параметри розміщення на формі
    pButton->Left = 10;
    pButton->Width = 100;
    pButton->Top = 10;
    pButton->Height = 20;
```

Цей код створює компонент типу TButton, розташований на формі в позиції 10,10 і протягнений на 100 пикселей вправо й 20 пикселей вниз. Подібний код може бути використаний для будь-якого типу компонентів, оскільки всі компоненти підтримують ці атрибути.

Якщо треба встановити інші властивості, то це робиться аналогічно через вказівник pButton. Якщо ж новостворений компонент повинен мати обробник деякої події, то заголовок майбутнього обробника цієї події треба додати в заголовочний файл форми у секцію private. Причому, прототип заголовку має повністю відповідати типу події.

Наприклад, для створеної кнопки вимоги для прототипу обробника події OnClick мають вигляд:

- один вхідний аргумент типу TObject* ;
- повертається тип void;
- обов'язково використання ключового слова __ fastcall.

Отже, прототип обробника події OnClick описується так:

```
private:
void __fastcall  OnButtonClick(TObject * Sender);
```

У файлі реалізації при створенні компонента додається рядок

```
pButton->OnClick=OnButtonClick;
```

А сам обробник має вигляд

```
void __fastcall TForm1::OnButtonClick(TObject* Sender)
{. . .}
```

Зазначимо, що є можливість переключення обробників подій під час роботи програми. Це робиться присвоюванням властивості pButton->OnClick імені іншого обробника подій.

Оскільки компонент Button був створений динамічно за допомогою оператора new, то і знищувати його треба за допомогою оператор delete, наприклад, так:

```
if(pButton){delete pButton; pButton=NULL;}
```

Автоматичне і динамічне створення форм

За замовчуванням при запуску додатка C++Builder автоматично створює по одному екземпляру кожного класу форм у проекті і звільняє їх при завершенні програми. Автоматичне створення обробляється C++Builder кодом, що генерується у трьох місцях.

Перше – інтерфейс класу форми у файлі із роширенням .h:

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

У даному фрагменті коду описується клас TForm1. Друге місце – це файл реалізації класу форми з розширенням .cpp.

```
TForm1 *Form1;
```

Тут описана змінна Form1, що вказує на екземпляр класу TForm1 і доступна з будь-якого модуля. Звичайно вона використовується під час роботи програми для керування формою.

Третє місце знаходиться в функції WinMain вихідного тексту проекту, доступ до якого можна одержати за допомогою меню **View | Project Source**. Цей код виглядає як:

```
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
. . .
Application->CreateForm(__classid(TForm1), &Form1);
. . .
}
```

Процес видалення форм обробляється за допомогою концепції власників об'єктів: коли об'єкт знищується, автоматично знищуються всі об'єкти, якими він володіє. Створена описаним вище способом форма належить об'єкту Application і знищується при закритті додатка.

Хоча автоматичне створення форм корисно при розробці SDI-додатків, при створенні MDI-додатку воно, як правило, неприйнятно. Для створення нового екземпляра форми використовується конструктор класу форми. Наведений нижче код створює новий екземпляр TForm1 під час роботи програми і встановлює його властивість Caption рівною “Нова форма”.

```
TForm1 * pForm1;
pForm1 = new TForm1 (Application);
pForm1->Caption = "Нова форма";
pForm1->Visible= true;
```

Конструктор одержує як параметр нащадка TComponent, що і буде власником форми. Звичайно в якості власника виступає Application, щоб усі форми були автоматично закриті по закінченні роботи додатка. Можна також передати конструктору параметр NULL, створивши форму без власника, але тоді закривати і знищувати її треба вручну.

Знищення форми відбувається за допомогою виклику методу Release() для неї. Це можна зробити, наприклад, так:

```
if(pForm1)pForm1->Release();
```

Створення багатодокументного інтерфейсу MDI

Термін MDI (Multiple Document Interface) дослівно означає багатодокументний інтерфейс. Він описує додатки, здатні завантажити і використовувати одночасно кілька документів чи об'єктів. Прикладом такого додатка може служити диспетчер файлів File Manager або текстовий редактор Microsoft Word.

Звичайно MDI-додатки складаються мінімум із двох форм – батьківської і дочірньої. Щоб форма була батьківською, властивість форми FormStyle установлюється рівною fsMDIForm. Для дочірньої форми стиль FormStyle встановлюється рівним fsMDIChild.

Батьківська форма служить контейнером, що містить дочірні форми, що укладені в клієнтську область і можуть переміщатися, змінювати розміри, мінімізуватися чи максимізуватися. У додатку можуть бути дочірні форми різних типів, наприклад одна – для обробки зображень, а інша – для роботи з текстом.

У MDI-додатку, як правило, потрібно створювати кілька екземплярів класів дочірньої форми. Оскільки кожна форма являє собою об'єкт, вона повинна бути створена перед використанням і звільнена, коли вона більше не потрібна. При створенні MDI-додатку екземпляри дочірньої форми створюються і знищуються динамічно.

Об'єкт типу TForm має кілька властивостей, методів і подій, специфічних для MDI-додатків.

MDI-властивості форми

ActiveMDIChild – ця властивість повертає дочірній об'єкт TForm, що має в поточний час фокус введення. Це корисно, коли батьківська форма містить панель інструментів чи меню, команди яких поширюються на відкриту дочірню форму.

MDIChildren і MDIChildCount. Властивість MDIChildren є масивом об'єктів TForm, що надає доступ до створених дочірніх форм. Властивість MDIChildCount повертає кількість елементів у масиві MDIChildren. Звичайно ця властивість використовується при виконанні якої-небудь дії над усіма відкритими дочірніми формами. Наприклад, код згортання всіх дочірніх форм деякою командою Minimize All може виглядати так:

```
void __fastcall
TMainForm::WindowMinimizeItemClick(TObject *Sender)
{
for(int iCount=MDIChildCount-1; iCount>=0; iCount--)
```

```
MDIChildren[iCount]->WindowState = wsMinimized;
}
```

TileMode – властивість типу, що визначає, як батьківська форма розміщає дочірні при виклику методу Tile. Використовуються значення tbHorizontal (за замовчуванням) і tbVertical для розміщення форм по горизонталі і вертикалі.

WindowMenu. Професійні MDI-додатки дозволяють активізувати необхідне дочірнє вікно, вибравши його зі списку в меню. Властивість WindowMenu визначає об'єкт TMenuItem, що C++Builder буде використовувати для виводу списку доступних дочірніх форм. Для виводу списку TMenuItem повинне бути меню верхнього рівня. Це меню має властивість Caption, рівне swindow.

MDI-методи форми

Специфічні для MDI-форм методи перелічено нижче.

- ***Arrangelcons*** вибудовує піктограми мінімізованих дочірніх форм у нижній частині батьківської форми;
- ***Cascade*** розташовує дочірні форм каскадом, так що видно всі їхні заголовки;
- ***Next*** і ***Previous*** переходить від однієї дочірньої форми до іншої в прямому і зворотному напрямі відповідно;
- ***Tile*** вибудовує дочірні форми так, що вони не перекриваються.

MDI-події

У MDI-додатку подія OnActivate запускається тільки при переключенні між дочірніми формами. Якщо фокус уведення передається з не MDI-форми в MDI-форму, генерується подія OnActivate батьківської форми, хоча її властивість Active ніколи і не встановлюється рівною true. Ця подія насправді строго логічна: адже, якби OnActivate генерувався тільки для дочірніх форм, не було б ніякої можливості довідатися про перехід фокуса уведення з іншого додатка.

Створення власних компонентів

Не дивлячись на те, що бібліотека VCL містить сотні компонентів та існують також пакети додаткових компонентів від різних розробників, іноді виникає необхідність створення власних компонентів або їх пакетів.

Схему створення нового компонента можна уявити так:

- вибір пращура майбутнього компонента;
- створення каркасу компонента з функцією реєстрації;
- добавлення властивостей, методів і подій;
- тестування компонента у власному проекті, використовуючи ще неінстальований динамічно створений компонент;
- інсталяція компонента на палітру;
- випробування інстальованого компонента.

Побудова каркасу компонента

Новий компонент краще всього спочатку створювати у власному проекті. Для цього треба створити новий проект і викликати майстра по створенню компонентів: **Component| New**.

У вікні, що з'явиться, треба назвати новий компонент і вибрати пращура, тобто клас, від якого буде породжений новий компонент. Нехай, в якості пращура обраний клас **TCustomMemo**. Назвемо новий компонент **TMyMemo** і виберемо місце розташування майбутнього компонента – сторінку **Samples** з палітри компонентів.

Після таких дій майстер по створенню компонентів згенерує два наведених нижче файли.

Файл MyMemo.h

```
// каркас компонента TMyMemo-нащадка TCustomMemo
class PACKAGE TMyMemo : public TCustomMemo
{
private:
protected:
public:
        __fastcall TMyMemo(TComponent* Owner);
__published:
};
```

Файл MyMemo.cpp

```
#include "MyMemo.h"
.   .   .
// функція перевірки наявності у компонента
// чистих віртуальних функцій
static inline void ValidCtrCheck(TMyMemo *)
{
        new TMyMemo(NULL);
}
```

```
//порожній конструктора компонента TMyMemo
__fastcall TMyMemo::TMyMemo(TComponent* Owner)
        : TCustomMemo(Owner)
{}

namespace Mymemo  // простір імен
{
void __fastcall PACKAGE Register()// функція реєстрації
    {
// масив компонентів, що реєструються
     TComponentClass classes[1] = {__classid(TMyMemo)};
// реєстрування компонентів
     RegisterComponents(
       "Samples", // ім'я вкладки палітри компонентів
        classes,  // масив компонентних класів
        0         // індекс останнього елемента масиву

                        );
    }
}
```

Тепер згенеровані файли (каркас компонента) треба наповнити кодами, тобто – додати необхідні властивості, методи і події.

Оголошення властивостей

Усі компоненти VCL дотримуються наступних угод про властивості:

- значення властивостей зберігають члени даних об'єкта;
- ідентифікатори членів даних, що зберігають значення властивостей, утворяться додаванням префікса F до імені цієї властивості. Так, наприклад, значення властивості Top компонента TControl зберігає член даних під іменем FTop.
- ідентифікатори членів даних, що зберігають значення властивостей, повинні бути оголошені як private. При цьому компонент, що оголосив ці властивості, має до них доступ, а користувач даного компонента і його похідних – немає.
- похідні компоненти повинні використовувати успадковану властивість, не намагаючись здійснити прямий доступ до пам'яті внутрішніх даних;
- тільки методи, що реалізують властивість, мають право доступу до своїх значень. Якщо якомусь іншому методу або компоненту

знадобиться змінити ці значення, вони повинні здійснювати це за допомогою даної властивості, а не звертаючись прямо до його внутрішніх даних.

C++Builder використовує ключове слово __property для оголошення властивостей [1, 28]. Синтаксис опису властивості має вигляд:

```
__property <тип властивості> <ім'я властивості> =
                                {<список атрибутів>};
```

де список атрибутів містить наступні атрибути властивості:

- write = < член даних або метод запису >;
- read = < член даних або метод читання >;
- default = < булева константа, що керує збереженням значення >;
- stored = < булева константа або функція, що зберігає значення >.

Існує два способи, за допомогою яких властивості забезпечують доступ до внутрішніх членів даних компонентів: прямий або непрямий за допомогою методів читання/запису.

Прямий доступ є найпростішим способом звертання до значень властивостей. Атрибути read і write вказують, що читання або присвоювання значень внутрішнім членам даних властивості відбувається безпосередньо, без виклику відповідних методів. Прямий доступ найчастіше використовується для читання значень властивостей.

Методи читання й запису заміщають імена членів даних в атрибутах read і write оголошення властивості. Вони повинні бути оголошені як приватні Оголошення методів приватними захищає користувача похідного компонента від випадкового виклику неадекватних методів, що модифікують властивості не так, як очікувалося.

Всі компоненти успадковують властивості своїх попередників, причому абстрактні базові класи зазвичай оголошують свої властивості у секціях з обмеженим доступом. Щоб такі властивості стали доступними користувачам похідних компонентів (як на стадії проектування, так і під час виконання програми), необхідно перевизначити їх з ключовим словом __published. У такому випадку

тип властивості не вказується, оскільки він визначений у батьківському класі.

Імена методів для читання властивостей прийнято починати префіксом Get, а імена методів для запису – префіксом Set. Назву самої властивості бажано використовувати як суфікс.

Значення властивості за замовчуванням встановлює конструктор даного компонента. C++Builder використовує оголошене значення властивості за замовчуванням default, щоб визначити, чи варто зберігати властивість у файлі форми з розширенням .dim (якщо атрибут stored явно не забороняє це).

Властивість може бути будь-якого типу, який здатна повернути функція. Різні типи властивостей по-різному представлені у вікні інспектора об'єктів і визначають різні варіанти їх редагування. Деякі властивості мають власні редактори.

Правилами мови C++ встановлюються наступні узагальнені групи типів компонентних властивостей [29]:

- ***Simple***. Прості числові, символьні і рядкові властивості показуються інспектором об’єктів у вигляді чисел, символів або символьних рядків відповідно. Можна безпосередньо вводити і редагувати значення простих властивостей.
- ***Enumerated.*** Властивості перелічуваного типу (зокрема булеві) показуються інспектором об’єктів у вигляді значень, визначених в початковому тексті програми. Можна вибирати можливі значення з випадного списку.
- ***Set.*** Властивості типу множини показуються інспектором об’єктів у вигляді елементів множини. При розширенні множини з кожним його елементом слід поводитися як з булевим значенням: true, якщо елемент належить множині, або false – інакше.
- ***Object.*** Властивості, які самі є об'єктами, зазвичай обслуговуються своїми власними редакторами. Інспектор об’єктів дозволяє індивідуально редагувати ті об'єктні властивості, які оголошені як _published. Об'єктні властивості повинні бути похідними від TPersistent
- ***Array.*** Властивості типу масив повинні обслуговуватися своїми власними редакторами властивостей. Інспектор об’єктів не має вбудованих засобів для редагування таких властивостей.

Для прикладу визначимо одну батьківську і одну власну властивість компонента TMyMemo:

```
class PACKAGE TMyMemo: public TCustomMemo
{
    . . .
    private: // закриті члени-дані класу
           bool FModified;
    public: // відкриті члени класу
        // власна властивість часу виконання
        __property bool Modified = {read=FModified,
                                    default=false};
    __published: //загальновідомі члени класу
           // батьківська властивість ScrollBars
           // стала видимою інспектору об\'єктів
            __property  ScrollBars;
};
```

Визначення подій

Формально C++Builder визначає подію як покажчик, що типізується, на метод в специфічному екземплярі класу:

```
<тип> (__closure *<ім'я методу>)(<список параметрів>);
```

Для розробника компонента closure є деякою програмною заглушкою: коли користувач визначає реакцію на деяку подію, місце заглушки займає його обробник, який викликається програмою користувача при виникненні цієї події.

Всі компоненти VCL успадковують більшість загальних повідомлень Windows – стандартні події. Такі події вбудовані в захищені секції компонентів, тому користувачі не можуть під'єднувати до них свої обробники.

При створенні нового компонента, щоб стандартні події були видимі інспекторові об'єктів на стадії проектування або під час виконання програми, треба перевизначити властивості події в секції public або __published.

```
class PACKAGE TMyMemo: public TCustomMemo
{
. . .
__published:
       // OnClick стало видимим інспектору
        __property  OnClick;
};
```

Тут тип властивості опущений, оскільки властивість є батьківською і її тип відомий.

Всі стандартні події мають відповідні захищені динамічні методи, успадковані від TControl, імена яких утворені від назви події без частинки "On". Наприклад, події OnClick викликають метод Click.

Необхідність у визначенні абсолютно нових подій виникає досить рідко. Зазвичай досить перевизначити обробку вже існуючих подій.

Якщо ж виникла необхідність створити власну подію, то треба оголосити тип і властивість для події. Якщо тип властивості описаний як __closure, то інспектор об'єктів визначає, що дана властивість є подією, і представляє її на вкладці events.

Нехай потрібно створити власну подію OnClear [1] з одним стандартним параметром типу TObject* і другим параметром булевого типу. Для цього треба оголосити новий тип події. Назвемо його, наприклад, TClear.

Оголошення робиться за допомогою ключового слова __closure. Тип TClear* можна визначити так:

```
typedef void __fastcall ( __closure *TClear)
        (System::TObject *Sender, bool CanClear);
```

Отже, визначено тип вказівника на функцію, що приймає два параметри: традиційний для всіх оброблювачів параметр Sender і параметр булевого типу CanClear (цей параметр користувач зможе змінювати).

Тепер нову подію можна визначити так:

```
class PACKAGE TMyMemo: public TCustomMemo
{
private :
        TClear FOnClear;
.   .   .
public:
        // власний метод
        virtual void __fastcall Clear();
.   .   .
__published:
        __property TClear OnClear = {read=FOnClear ,
                                     write=FOnClear};
};
```

Оскільки подія OnClear викликає метод Clear, то необхідно змінити метод Clear так, щоб у ньому враховувався новий параметр CanClear:

```
void __fastcall TMyMemo::Clear()
{
 bool CanClear = true;
 if (OnClear) OnClear (this, CanClear);
 if (CanClear)
  {
   TCustomMemo::Clear();// виклик батьківського методу
   FModified = false;
  }
}
```

У цій функції вводиться булева змінна CanClear. При наявності обробника користувача він викликається й CanClose передається в цей обробник другим параметром. Потім, залежно від значення CanClose, викликається або не викликається батьківський метод очищення Clear.

Додавання методів

Компонентні методи нічим не відрізняються від інших функцій-членів класу. Всі методи (включаючи конструктори), до яких мають право звертатися користувачі компонента, слід оголошувати як public. Всі методи, до яких мають право звертатися похідні об'єкти компоненту, слід оголошувати як protected. Це забороняє користувачам передчасно викликати методи, дані для яких ще не створені.

Якщо треба не додати новий метод у компонент, а перевизначити метод з батьківського класу, то необхідно звернутися до довідки C++Builder і точно відтворити у новому класі оголошення перевизначеного методу (скопіювати його прототип).

Наприклад, для компонента TMyMemo треба перевизначити метод ClearSelection, що очищає текст, виділений у вікні редагування. Оголошення методу ClearSelection у класі-предку TCustomMemo має вигляд:

```
void __fastcall ClearSelection(void);
```

Отже, оголошення й реалізація перевизначеного методу ClearSelection може мати вигляд:

```
class PACKAGE TMyMemo: public TCustomMemo
{
.  .  .
 public:
    void __fastcall ClearSelection(void);
};

void __fastcall TMyMemo::ClearSelection()
{
.  .  .    // власний код
// виклик батьківського метода
TCustomMemo::ClearSelection();
}
```

Розробка графічних додатків

VCL – надбудова над GDI

GDI (Graphics Device Interface) – це та частина Windows, що забезпечує підтримку апаратно-незалежної графіки. C++Builder інкапсулює функції Windows GDI на різних рівнях [1]. Найбільш важливим тут є спосіб, за допомогою якого графічні компоненти представляють свої зображення на екрані монітора. При прямому виклику функції GDI необхідно передавати їм дескриптор контексту пристрою (device context handle), що задає обрані інструменти малювання – перо, пензель і шрифт. Після завершення роботи із графічними зображеннями обов'язково треба привести контекст пристрою у вихідний стан і тільки потім звільнитися від нього.

Типовий приклад малювання із застосовуванням стандартних функцій GDI може виглядати так:

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
HDC dc = GetDC(Handle); // дескриптор контексту
HPEN PenHandle, OldPenHandle;
PenHandle = CreatePen(PS_SOLID, 1, RGB(255, 0, 0));
OldPenHandle = SelectObject(dc, PenHandle);

HBRUSH BrushHandle, OldBrushHandle;
BrushHandle = CreateSolidBrush(RGB(255, 255, 0));
```

```
OldBrushHandle = SelectObject(dc, BrushHandle);

// жовтий еліпс, обведений червоним контуром
Ellipse(dc, 10, 10, 120, 100);

SelectObject(dc,OldBrushHandle) ;
DeleteObject(BrushHandle) ;
SelectObject(dc,OldPenHandle);
DeleteObject(PenHandle) ;
}
```

Замість того, щоб працювати із графікою на такому рівні деталізації, C++Builder надає простий і завершений інтерфейс за допомогою властивості Canvas (канва) графічних компонентів бібліотеки VCL. Ця властивість ініціалізує правильний контекст пристрою й звільняє його в потрібний час, коли припиняється малювання. Дескриптор контексту пристрою, над яким "побудована" канва, може бути потрібним для різних низькорівневих операцій. Він задається властивістю канви Handle. Клас TCanvas є обгорткою для HDC (HDC доступний через властивість Handle) і представляє більш високорівневий інтерфейс для роботи із графікою.

За аналогією з функціями Windows GDI канва має вкладені властивості, що представляють характеристики пера (Pen), пензеля (Brush) й шрифту (Font).

Єдине, що повинен зробити користувач, працюючи із графічними компонентами, – це визначити характеристики використовуваних інструментів малювання. З використанням Canvas зникає необхідність стежити за системними ресурсами при створенні, виборі й звільненні інструментів. Канва сама дбає про це.

У ряду компонентів з бібліотеки візуальних компонентів VCL є властивість Canvas (канва), яка надає простий шлях для малювання на них. TCanvas – це об'єктний клас, який інкапсулює графічні функції Windows. Канва не є компонентом, але вона входить як властивість у ряд компонентів, які повинні вміти намалювати себе й відобразити яку-небудь інформацію. Canvas (канва) надає простий шлях для малювання, наприклад, на компонентах TImage, TPaintBox, TForm.

Канва містить методи-надбудови над всіма основними функціями малювання GDI Windows. Всі геометричні фігури малюються поточним пером. Ті з них, які можна зафарбовувати, зафарбовуються за допомогою поточного пензля. Пензель і перо при цьому мають

поточний колір. Крім того, можна малювати й поточково, одержавши доступ до кожного пикселя.

Клас TCanvas інкапсулює графічні методи: Draw, TextOut, Arc, Rectangle та ін. Використовуючи властивість Canvas, можна відтворювати на формі будь-які графічні об'єкти – картинки, многокутники, текст і т.п. без використання компонентів TImage, TShape і TLabel (тобто без використання додаткових ресурсів), однак при цьому треба обробляти подію OnPaint того об'єкта, на канві якого відбувається малювання.

Розглянемо приклад коду з використанням Canvas, який знову, як і в попередньому прикладі, малює жовтий еліпс, обведений червоним контуром, і наочно ілюструє, наскільки C++Builder спрощує програмування графіки.

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
Canvas->Pen->Color = clRed; // колір контуру
Canvas->Brush->Color = clYellow; // колір заливки

// жовтий еліпс, обведений червоним контуром
Canvas->Ellipse(10, 10, 120, 100);
}
```

Тут відбувалось малювання прямо на формі (можна було це вказати явно Form1-> Canvas->Ellipse(10, 10, 120, 100);).

Зазначимо, що тільки частина компонентів має властивість Canvas. Якщо треба малювати на компоненті, який не має такої властивості (наприклад, TPanel, TButton та інші), то здавалося б, що немає іншого виходу, ніж із застосовуванням стандартних функцій GDI. Але, виявляється, що за допомогою класу TControlCanvas можна приєднати канву до компонента, у якого її немає, і далі просто малювати так, як завжди.

Продемонструємо малювання еліпса на панелі (яка не має властивості Canvas) при натискання командної кнопки ***Button1***:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
TControlCanvas *p=new TControlCanvas();
p->Control=(TControl*)Panel1;
p->Ellipse(10,10,120,100);
}
```

Об'єктний клас TCanvas

Canvas зв'язаний з усіма компонентами VCL, у яких є клієнтська частина, а також із класом TBitmap. Стандартні компоненти Windows такі як кнопки, списки й т.д. не мають властивості Canvas, тому що їх повністю малює Windows. Малювання на канві відбувається шляхом виклику відповідних функцій-членів (методів Draw, TextOut, Arc, Rectangle, Ellipse та ін.), а переключення режимів малювання відбувається шляхом модифікації властивостей класу TCanvas – Pen, Font, Brush, TextFlags та інших.

Основні властивості класу TCanvas

Brush – пензель, є об'єктом зі своїм набором властивостей:

Bitmap - картинка розміром строго 8×8, використовується для заповнення (заливання) області на екрані;

Color - колір заливання;

Style - визначений стиль заливання; ця властивість конкурує з властивістю Bitmap (та властивість з них, яка встановлена останньою, і буде визначати вигляд заливання;

Handle - дана властивість дає можливість використовувати пензель у прямих викликах процедур Windows API .

ClipRect – (тільки читання) прямокутник, на якому відбувається графічний вивід.

CopyMode – властивість визначає, яким чином буде відбуватися копіювання (метод CopyRect) на дану канву зображення з іншого місця: один до одного, з інверсією зображення й ін.

Font – шрифт, яким виводиться текст (методом TextOut).

Handle – використовується для прямих викликів Windows API.

Pen – перо, визначає вид ліній; як і пензель (Brush) є об'єктом з набором властивостей:

Color - колір лінії;

Handle - для прямих викликів Windows API;

Mode - режим виводу: проста лінія, з інвертуванням, з

виконанням виключаючого або та інші;

Style - стиль виводу: лінія, пунктир та інші;

Width - ширина лінії в точках;

PenPos - поточна позиція пера, перо рекомендується переміщати за допомогою методу MoveTo, а не прямою установкою даної властивості;

Pixels - двомірний масив елементів зображення (pixel), з його допомогою одержується доступ до кожної окремої точки зображення.

Основні методи класу TCanvas

- Методи для малювання найпростішої графіки:

MoveTo – встановлює поточну позицію пера (PenPos);
LineTo – малює пряму до заданої точки;
Rectangle – малює прямокутник із заданою діагоналлю;
Ellipse – малює еліпс, вписаний в заданий діагоналлю прямокутник;
Arc – малює частину кривою еліпсу;
Chord– малює частину кривою еліпсу, з'єднану хордою;
Pie – малює сектор еліпсу;
Polygon –малює замкнуту ламану;
PolyLine –малює незамкнуту ламану;
RoundRect – малює заокруглений прямокутник.

При промальовуванні ліній у цих методах використовуються перо (Pen) канви, а для заповнення внутрішніх областей – пензель (Brush).

- Методи для виводу тексту:

TextOut – виводить текстовий рядок (шрифтом (Font) канви);
TextRect – виводить текстовий рядок в прямокутнику (використовується шрифт (Font) канви);
TextHeight – задає його висоту;
TextWidth – задає його ширину.

- Методи для виводу картинок на канву:

Draw. Як параметри вказуються прямокутник і графічний об'єкт для виводу (це може бути TBitmap, TIcon або TMetafile);

StretchDraw. Від *Draw* відрізняється тим, що розтягує або стискає картинку так, щоб вона заповнила весь зазначений прямокутник.

- Методи для замальовування областей:

FillRect – замальовує прямокутник кольором і стилем пензля;
FloodFill – замальовує область довільної форми кольором і стилем пензля.

Подія OnPaint

При малюванні на канві форми або по **PaintBox** треба враховувати деякі особливості. Якщо вікно якогось іншого додатка перекриває вікно вашого додатка або, якщо малюнок тимчасово згортається, то зображення, намальоване на канві форми, псується. У компоненті **Image** цього не відбувається, оскільки в класі **TImage** уже передбачені всі необхідні дії, що здійснюють перемальовування зіпсованого зображення. А при малюванні на канві форми або інших віконних компонентів перемальовуванням повинен займатися сам розроблювач додатка. Якщо вікно було перекрито й зображення зіпсувалося, операційна система повідомляє додатку, що в оточенні щось змінилося й що додаток повинен почати відповідні дії. Якщо потрібне відновлення вікна, для нього генерується подія **OnPaint** [1]. В обробнику цієї події потрібно перемалювати зображення. Перемальовування може відбуватися різними способами залежно від задачі.

Припустимо на формі треба розмістити малюнок з деякого графічного файлу. Це можна зробити, помістивши у секцію private інтерфейсної частини класу TForm1 (файл Unit1.h) рядок:

```
//  вказівник на графічний об'єкт з класу Graphics
Graphics:: TBitmap *pBitmap;
```

а у тіло конструктора класу TForm1 (файл Unit1.cpp) – рядок:

```
pBitmap = new Graphics:: TBitmap;
```

```
// обробник події OnClick малювання картинки
// (файл Unit1.cpp)
void __fastcall TForm1::FormClick(TObject *Sender)
{
if (OpenPictureDialog1->Execute())
pBitmap->LoadFromFile(OpenPictureDialog1->FileName);
Canvas->Draw(0,0,pBitmap);
}
```

Оскільки графічний об'єкт створився динамічно (оператор new), треба його знищити за допомогою оператора delete. Краще всього це зробити за допомогою деструктора класу (секція public).

Файл Unit1.h:

```
__fastcall ~TForm1 (void);
```

Файл Unit1.cpp:

```
__fastcall ~TForm1::TForm1 (void)
{
delete pBitmap;
}

// обробник події OnPaint (файл Unit1.cpp)
void __fastcall TForm1::FormPaint(TObject *Sender)
{
if (pBitmap != NULL) Canvas->Draw(0,0,pBitmap);
}
```

Наведений вище обробник події **OnPaint** перемальовує все зображення, хоча, може бути, зіпсована тільки частина його. При великих зображеннях це може значно уповільнювати перемальовування. Перемальовування можна істотно прискорити, якщо перемальовувати тільки зіпсовану область канви.

У канви є властивість **ClipRect** типу TRect, що у момент обробки події **OnPaint** указує на область, що підлягає перемальовуванню. Тому більше ефективним буде обробник:

```
void_fastcall TForm1::FormPaint(TObject *Sender)
 {
 if (pBitmap != NULL)Canvas->CopyRect
```

```
(Canvas->ClipRect, pBitmap->Canvas, Canvas->ClipRect);
}
```

Він перемальовує тільки прямокутну область **ClipRect,** яка зіпсована.

Візуальна розробка додатків баз даних

Механізми доступу до даних із засобів розробки

Існує кілька способів доступу до даних із засобів розробки і клієнтських додатків [22].

Більшість систем керування базами даних містить у своєму складі бібліотеки, що надають спеціальний ***прикладний програмний інтерфейс*** *(Application Programming Interface, API)* для доступу до даних цієї СКБД.

У цьому випадку створений додаток зможе використовувати дані тільки СКБД цього виробника, і заміна її на іншу (наприклад, з метою розширення сховища даних або переходу в архітектуру "клієнт-сервер") призведе до переписування значної частини коду клієнтського додатка. Клієнтські API і об'єктні моделі не підкоряються ніяким стандартам і різні для різних СКБД.

Інший спосіб маніпуляції даними в додатку базується на застосуванні ***універсальних механізмів*** доступу до даних. Універсальний механізм доступу до даних звичайно реалізований у вигляді бібліотек і додаткових модулів, називаних ***драйверами*** або ***провайдерами***, що нерідко підкоряється тій або іншої специфікації. Додаткові модулі, специфічні для тієї або іншої СКБД, реалізують безпосереднє звертання до функцій клієнтського API конкретних СКБД.

Найбільш популярними серед універсальних механізмів доступу до даних є наступні:

- Open Database Connectivity (ODBC).
- OLE DB.
- Active Data Objects (ADO).
- Borland Database Engine (BDE).

Універсальні механізми ODBC, OLE DB і ADO фірми Microsoft являють собою власне кажучи промислові стандарти. Що стосується

механізму доступу до даних BDE фірми Borland, то він так і не став промисловим стандартом, однак донедавна застосовувався досить широко, тому що до виходу Delphi 5 і C++Builder 5 був практично єдиним універсальним механізмом доступу до даних, підтримуваним засобами розробки Borland на рівні компонентів і класів.

У загальному випадку додаток, що використовує бази даних, може застосовувати, наприклад, наступні механізми доступу до них:

- безпосередній виклик функцій клієнтського API (або звертання до COM-об'єктів клієнтських бібліотек);
- виклик функцій ODBC API (або застосування класів, що інкапсулюють подібні виклики);
- безпосереднє звертання до інтерфейсів OLE DB;
- застосування ADO (або застосування класів, що інкапсулюють звертання до об'єктів ADO);
- застосування ADO + OLE DB + ODBC;
- застосування BDE + SQL Links (або застосування класів, що інкапсулюють звертання до функцій BDE);
- застосування BDE + ODBC Link + ODBC.

Крім цих існують і інші способи доступу до даних.

Використовуючи C++Builder, можна створювати додатки, які працюють як з ***автономними*** (автономні локальні бази даних зберігають свої дані в локальній файловій системі на тому же комп'ютері, що і СКБД, мережа не використовується), так і з ***серверними*** СКБД, такими як Oracle, Sybase, Interbase, MS SQL, Informix, DB2, а також з ODCB-джерелами (Open Database Connectivity).

Робота з даними в C++Builder в основному виконується через **Borland Database Engine** (BDE) процесор баз даних фірми Borland. Відповідна програма має бути встановлена на комп'ютері користувача. BDE є посередником між додатком та базами даних. Він надає користувачу єдиний інтерфейс для роботи, що звільняє від необхідності знати конкретну реалізації бази даних. Завдяки цьому, не треба змінювати додаток при зміні реалізації бази даних. Додаток C++Builder звертається до бази даних через BDE.

Починаючи з C++Builder 5, введена інша альтернативна можливість роботи з базами даних, минаючи BDE. Це – розроблена в Microsoft технологія **ActiveX Data Objects** (ADO). ADO – це користувацький інтерфейс до будь-яких типів даних, включаючи бази

даних, електронну пошту системні, текстові і графічні файли, Зв'язок з даними здійснюється засобами так званої технології OLE DB.

Ще один альтернативний доступ до баз даних Interbase був введений у C++Builder 5 на основі технології **InterBaseExpress** (IBX). У бібліотеці компонентів C++Builder 5 з'явилася сторінка *InterBase*, що містить компоненти роботи з InterBase, минаючи BDE. Ці компоненти забезпечують підвищену працездатність і дозволяють використовувати нові можливості сервера InterBase, недоступні звичайним компонентам BDE.

У C++Builder 6 введена ще одна нова технологія доступу до баз даних – **dbExpress.** Це драйвери, що забезпечують доступ до серверів SQL на основі єдиного інтерфейсу. При використанні dbExpress можна поставляти свій додаток разом з DLL відповідного драйвера й, якщо треба, з додатковими файлами, що містять інформацію про з'єднання.

У C++Builder 6 також розширені можливості побудови клієнтських додатків за рахунок появи нових компонентів BDEClientDataSet, SQLClientDataSet, IBClientDataSet і розширення можливостей компонента ClientDataSet. Нові компоненти SharedConnection і LocalConnection дозволяють створювати істотно більш гнучкі багатопоточні додатки із клієнтськими наборами даних.

У C++Builder 6 є окремі сторінки BDE, ADO, InterBase, dbExpress, які містять компоненти, що забезпечують організацію відповідних технологій.

При створенні додатків баз даних зручно не просто вказувати шлях доступу до таблиць бази даних, а використовувати для цього так званий ***псевдонім (аліас)***. Він зберігається в окремому конфігураційному файлі у довільному місці на диску і дозволяє виключити з програми пряму вказівку шляхів доступу до бази даних. Такий підхід дає можливість розташовувати дані в будь-якому місці, не перекомпільовуючи при цьому програму. Крім шляху доступу, в аліасі вказуються тип бази даних, драйвер і багато іншої керуючої інформації. Тому використання аліасів дозволяє легко переходити, наприклад, від локальних баз даних до SQL-серверних баз при виконанні вимог поділу додатка на клієнтську і серверну частини.

Універсальний механізм доступу ODBC

ODBC (Open Database Connectivity) – широко розповсюджений програмний інтерфейс фірми Microsoft, що задовольняє стандартам ANSI і ISO для інтерфейсів звертань до баз даних (Call Level Interface, CLI) [22]. Для доступу до даних конкретної СКБД за допомогою

ODBC, крім власне клієнтської частини цієї СКБД потрібний ODBC Administrator (додаток, що дозволяє визначити, які джерела даних доступні для даного комп'ютера за допомогою ODBC, і описати нові джерела даних), і ODBC-драйвер для доступу до цієї СКБД. ODBC-драйвер являє собою бібліотеку, що завантажується динамічно (DLL), яку клієнтський додаток може завантажити у свій адресний простір і використовувати для доступу до джерела даних. Для кожної використовуваної СКБД потрібний власний ODBC-драйвер, тому що ODBC-драйвери використовують функції клієнтських API, різні для різних СКБД.

За допомогою ODBC можна маніпулювати даними будь-якої СКБД (і навіть даними, що не мають прямого відношення до баз даних, наприклад, даними у файлах електронних таблиць або в текстових файлах), якщо для них є ODBC-драйвер. Для маніпуляції даними можна використовувати як безпосередні виклики ODBC API, так і інші універсальні механізми доступу до даних, наприклад OLE DB, ADO, BDE, що реалізують стандартні функції або класи на основі викликів ODBC API у драйверах або провайдерах, спеціально призначених для роботи з будь-якими ODBC-джерелами.

Доступ до баз даних через BDE

BDE (Borland Database Engine) – універсальний механізм доступу до даних, що застосовується у засобах розробки фірми Borland (а саме – Delphi і C++Builder), а також у деяких інших продуктах, наприклад Corel Paradox, Corel Quattro Pro, Seagate Software Crystal Reports [22].

BDE – це спадкоємець бібліотеки Paradox Engine, створеної для Borland Pascal і Borland C++ з метою надати додаткам, розробленим за їхньою допомогою, доступ до таблиць СКБД Paradox. Незабаром після створення Paradox Engine компанією Borland було розроблено кілька бібліотек-драйверів під загальною назвою SQL Links. Ці бібліотеки розширили функціональність BDE, дозволивши застосовувати набір функцій, який був в Paradox Engine, для доступу до даних dBase, ODBC-джерел, а також найбільш популярних серверних СКБД. Пізніше до цього набору були додані бібліотеки для доступу до Access і FoxPro.

Фізично BDE являє собою набір бібліотек доступу до даних, що реалізують BDE API – набір функцій для маніпуляції даними, що викликаються з додатка. Ці функції, у свою чергу, можуть звертатися до функцій клієнтського API (у випадку, наприклад, Oracle, Informix,

IB Database) або ODBC API (Access 2000, Microsoft SQL Server 7.0, будь-які ODBC-джерела), а також безпосередньо маніпулювати файлами деяких СКБД (dBase, Paradox).

Для доступу до бази даних за допомогою BDE на комп'ютері, що містить клієнтський додаток, повинні бути встановлені бібліотеки BDE загального призначення, а також BDE-драйвер для даної СКБД. У випадку серверних СКБД такі драйвери звуться SQL-Links. Ці драйвери містять BDE API – стандартний набір функцій, створених на основі функцій клієнтських API відповідних СКБД.

Серед BDE-драйверів є драйвер, створений з використанням ODBC API, – так званий ODBC Link, що застосовується разом з ODBC-драйвером для обраної СКБД.

Для доступу за допомогою BDE до Access 2000 можна використовувати відповідний ODBC-драйвер і ODBC Link, при цьому на комп'ютері, де експлуатується додаток, що їх використовує, потрібна наявність Microsoft Jet Engine 4.0. Він входить до складу Microsoft Access 2000, а також до складу Microsoft Data Access Components (MDAC).

У C++Builder доступ до баз даних за допомогою BDE можна організувати, наприклад, через компонент TDatabase, вказавши в ньому псевдонім, а доступ до кожної таблиці відбувається через компоненту типу TTable (їх може бути багато, вони можуть бути пов’язані). В свою чергу зв’язок між таблицями і елементами керування (наприклад, типу TDBGrid) здійснюється через компонент TDataSource). Ці три компонента знаходяться на сторінках Data Access (TDataSource), BDE, Data Control. Їх можна помістити в один модуль даних – компонент TDataModule.

Зазначимо, що використання BDE – не найефективніший спосіб доступу до даних Access. Більш доцільно здійснювати доступ до даних Access за допомогою ADO і OLE DB, тому що ці механізми надають у порівнянні з BDE набагато більше функціональних можливостей.

Доступ до баз даних через ADO

Як було сказано вище, починаючи з C++Builder 5, з’явилася можливість роботи з базами даних засобом розробленої в Microsoft технології ActiveX Data Object (ADO). ADO – це користувацький інтерфейс до любих типів даних, включаючи реляційні та нереляційні бази даних, електронну пошту, системні, текстові і графічні файли. Зв'язок з даними здійснюється засобом так званої технології OLE DB.

Використання ADO являється альтернативою Borland Database Engine (BDE), яка забезпечує більш ефективне роботу з даними. Для використання цієї можливості на комп'ютері повинна бути встановлена система ADO (міститься в останніх версіях Windows). Крім того, повинна бути встановлена клієнтська система доступу до даних, наприклад, Microsoft SQL Server, а в ODBC повинен міститися драйвер OLE DB для того типу баз даних, з яким іде робота.

Для роботи з ADO в C++Builder передбачені компоненти, розміщенні на сторінці бібліотеки – ADO. Вони інкапсулюють такі об'єкти ADO, як Connection, Command і Recordset. Їм відповідають компоненти C++Builder TADOConnection, TADOCommand і TADODataSet.

Зв'язок з базою даних в технології ADO здійснюється звичайним ланцюжком: набір даних => джерело даних (компонент TDataSourse) => компоненти управління і відображення даних (TDBGrid, TDBEdit та ін.).

Перелічимо далі основні компоненти зі сторінки ADO палітри компонентів.

TADOConnection	використовується для зв'язку з набором даних ADO. Може працювати з кількома компонентами наборів даних як диспетчер виконання їх команд
TADODataSet	універсальний компонент зв'язку з наборами даних, який може працювати в різних режимах, заміняючи зв'язані з BDE компоненти TTable, TQuery, TStoredProc
TADOTable	використовується для роботи з одною таблицею. Може зв'язуватись з нею безпосередньо або через TADOConnection
TADOQuery	використовується для роботи з набором даних за допомогою запитів SQL, включаючи такі запити мови DDL, як CREATE TABLE
TADOStoredProc	використовується для виконання процедур, які зберігаються на сервері
TADOCommand	використовується в основному для виконання команд SQL, які не повертають множину результатів

Доступ до бази даних здійснюється або за допомогою рядка з'єднання – властивості ConnectionString, або за допомогою окремого

компонента TADOConnection, ім'я якого задається у властивості Connection інших компонентів.

Взаємодія компонентів доступу до даних з інтерфейсними компонентами

Компоненти доступу до даних (data access control) містяться на сторінці палітри компонентів Data Access, BDE, ADO, InterBase, dbExpress. Вони забезпечують додатку доступ до баз даних, невидимі під час виконання програми (невізуальні). До них відносяться, наприклад, TDatabase, TADOTable, TTable,TDataSource, TQuery, TADOQuery.

Інтерфейсний елемент (data-aware control) – візуальний (видимий під час виконання програми) компонент, який дозволяє користувачу переглядати і змінювати дані в базі, використовуючи компоненти доступу до даних. Інтерфейсні компоненти знаходяться на вкладинці Data Controls палітри компонентів.

TDBGrid	використовується для табличного відображення даних
TDBNavigator	це набір зі звичайних кнопок розташованих на єдиній панелі, дозволяють маніпулювати положенням курсору, додавання рядків, видалення й т.д.
TDBText	є аналогом компонента TLabel, але дозволяє встановлювати текст із певного стовпця БД
TDBEdit	є аналогом компонента TEdit (дозволяє демонструвати і вводити дані стовпця БД вручну)
TDBMemo	аналог багаторядкового редактора тексту
TDBImage	дозволяє відображати графічні зображення, що утримуються у файлі бази даних
TDBListBox	дозволяє вибирати значення зі списку наявних значень
TDBComboBox	є поєднанням TDBEdit і TDBListBox, тільки використовується спадаючий список
TDBCheckBox	активізує/деактивізує одне значення
TDBRadioGroup	сукупність компонентів (аналог TRadioButton)

TDBLookUpList	список значень, підібраних з іншого стовпця або іншої таблиці
TDBLookUpComboBox	аналогічно TDBLookUpList
TDBRichEdit	аналогічно TDBMemo, але має можливість більше детального відображення й редагування (колір, розмір)
TDBCtrlGrid	відображає записи таблиці
TDBChart	компонент використовується для побудови одно- і багато- серійних графіків

C++Builder підтримує "триступінчасту" модель розробки додатка баз даних. У цій моделі компонент керування зв'язаний з компонентом джерела, а той, у свою чергу, одержує фактичні дані таблиці або запиту.

Набір даних у C++ Builder – це об'єкт, що складається з набору записів, кожен з яких, у свою чергу, складається з полів, і покажчика поточного запису. Набір даних може мати повну відповідність з реально існуючою таблицею або бути результатом запиту, він може бути частиною таблиці або поєднувати між собою кілька таблиць.

Набір даних у C++Builder є нащадком абстрактного класу TDataSet Наприклад, класи TQuery, TTable і TStoredProc, що містяться на сторінці палітри компонентів Data Access, – спадкоємці TDBDataSet, що, у свою чергу, є спадкоємцем TDataSet. TDataSet містить абстракції, необхідні для безпосереднього керування таблицями або запитами, забезпечуючи засоби для того, щоб відкрити таблицю або виконати запит і переміщатися по рядках.

TTable, TADOTable – компоненти C++Builder, які забезпечують доступ до таблиць баз даних. Властивість TableName цих компонентів використовується для вибору конкретної таблиці, до якої потрібно звернутися.

TField – клас C++Builder, який забезпечує доступ до полів таблиці бази даних.

TQuery, TADOQuery – компоненти C++Builder, які дозволяють створювати і обробляти власні запити SQL.

TStoredProc – компонент C++Builder, який дозволяє запускати відкомпільовані процедури SQL, які знаходяться на сервері баз даних. Такі процедури називаються процедурами, що зберігаються (stored procedures).

TDataSource (сторінка Data Access) – компонент C++Builder, який забезпечує зв’язок компонентів, нащадків TDataSet, з інтерфейсними

компонентами. Інтерфейсні елементи звертаються до компонентів TDataSource, які в свою чергу звертаються до компонентів доступу до даних (data access control) – нащадків TDataSet.

У додатку інтерфейсні компоненти звертаються до компонентів типу TDataSource. Як правило, в формі міститься невелика кількість компонентів типу TDataSource, проте інтерфейсних компонентів може бути багато. Компоненти TDataSource, у свою чергу, звертаються до одного або декількох компонентів, нащадків TDataSet.

У додатку не обов'язково використовувати компонент TDatabase для звертання до бази даних. Він надає деякі додаткові можливості, які не обов'язково використовувати, але він не є невід'ємною частиною додатків C++Builder, які працюють з базами даних. Нижче показаний можливий взаємозв'язок названих компонентів у випадку використання компонентів доступу до баз даних з вкладники BDE.

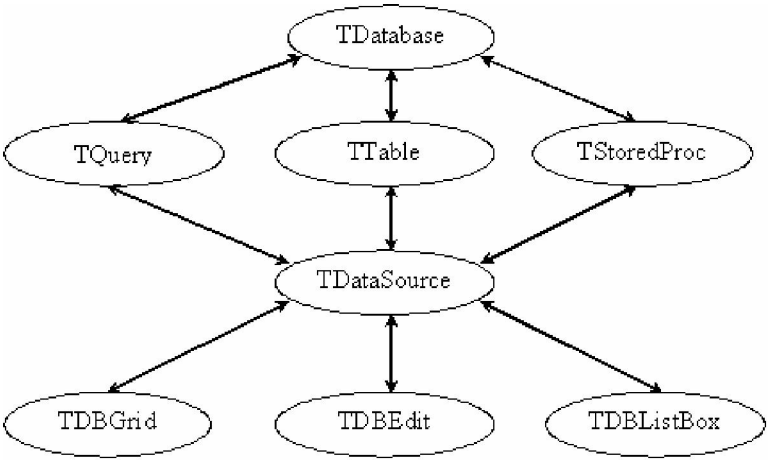

Рис. 2.

Етапи проектування додатка СКБД у найпростішому випадку

Нехай треба створити простий додаток СКБД з використанням для доступу до бази даних псевдоніму (доступ здійснювати через ADO або BDE), для компонентів доступу до даних – модуля даних, для інтерфейсної частини проекту – окремої форми з елементами відображення даних.

Процес створення такого додатку можна умовно поділити на п’ять етапів.

1. Налагодження джерел даних ODBC з використанням для цього *псевдоніма (аліаса)*.

Псевдонім (alias) містить всю інформацію, необхідну для забезпечення доступу до бази даних. Ця інформація повідомляється тільки один раз при створенні псевдоніма. А додаток для зв'язку з базою даних використовує псевдонім. У цьому випадку додатку байдуже, де фізично міститься та чи інша база даних, а часто байдужа і СКБД, що створила і обслуговує цю базу даних. При зміні системи каталогів, сервера тощо. в додатку нічого переробляти не треба. Досить, щоб адміністратор ввів відповідну інформацію у псевдонім.

Створити псевдонім до бази даних можна через адміністратор джерел даних ODBC. Для цього у панелі управління треба вибрати ***Администрирование | Источники данных ODBC*** (для Windows XP), далі вибрати сторінку ***Пользовательский (User DSN)*** і додати нове джерело даних, натиснувши кнопку ***Добавить (Add)*** і вибравши у вікні, що відкриється, потрібний драйвер, наприклад, ***Microsoft Access Drives***. Після цього з’явиться наступне вікно, наприклад, ***ODBC Microsoft Access Setup*** з полем вводу ***Data Source Name***. Тут можна ввести ім’я псевдоніму, а за допомогою кнопки ***Select*** – вибрати базу даних, яка буде відповідати створеному псевдоніму.

2. Перенесення на спеціалізовану форму (модуль даних) компонента набору даних з вкладинки BDE (TTable або TQuery) або з вкладинки ADO (TADOTable або TADOQuery) із сторіки палітри Data Access і встановлення його властивостей у відповідності зі специфічними вимогами обраної бази даних (вказується, наприклад, псевдонім).

3. Перенесення на форму (модуль даних) компонента джерела даних TDataSource з вкладинки Data Access. При цьому у властивості DataSet указується властивість Name об'єкта набору даних з п.2 (наприклад, Table1 або Query1).

4. Перенесення на форму (інтерфейсна частина проекту) потрібних компонентів відображення і редагування даних із вкладки Data Controls. При цьому у властивості DataSource задається джерело

даних (наприклад, DataSourcel). Відображуване поле набору даних вказується у властивості DataField.

5. Якщо на попередньому кроці обрано компонент сітка TDBGrid (тоді властивість DataField не потрібна), то треба додати на форму навігатор TDBNavigator, який використовується разом з компонентом TDBGrid.

Лабораторні роботи

Лабораторна робота №1

Тема: Використання стандартної бібліотеки шаблонів STL

Побудувати проект SDI з використанням стандартної бібліотеки шаблонів STL.

Для цього

- вхідні дані з елементів керування форми занести у зручні для реалізації задачі контейнери, утворені засобами бібліотеки шаблонів STL (не обов'язково використовувати саме запропоновані у варіантах завдань контейнери);
- при розв'язуванні задачі використати ітератори і алгоритми бібліотеки STL;
- передбачити три командні кнопки, по яких елементи керування форми заповнюються підготовленими тестовими прикладами.

Інтерфейс програми повинен бути зручним і зрозумілим довільному користувачу. У програмі при можливості передбачити контроль вводу користувача. Елементи керування, які тимчасово не використовуються, зробити недоступними або невидимими.

Варіанти завдань

1. Детермінований скінченний автомат вводиться за допомогою текстової таблиці TStringGrid. Зчитати його у пам'ять. Представлення в пам'яті – у вигляді масиву з елементами вигляду

стан, ознака заключності, символ переходу, стан після переходу

Для реалізації використати зручний контейнер, можливо, такий – *vector<pair<pair<char,boolean>, <pair<char,char> >*. За допомогою

утвореного представлення визначити, чи належить введений ланцюжок мові автомата. Вивести отриману таблицю переходів у багаторядкове поле редагування TMemo.

2. Недетермінований скінченний автомат вводиться за допомогою TStringGrid. Зчитати його у пам'ять. Представлення у пам'яті – у вигляді масиву з елементами вигляду

стан, символ переходу, рядок станів після переходу

Для реалізації використати зручний контейнер, можливо, такий – *vector<pair<pair<char,char>,string>*. Вивести отриману таблицю переходів у багаторядкове поле редагування TMemo. За допомогою утвореного представлення реалізувати алгоритм вилучення недосяжних станів автомата.

3. У багаторядковому полі редагування TMemo задано праволінійну граматику. Кожному нетерміналу граматики відповідає один рядок TMemo. Правила у рядку розділяються символом "|". Зчитати граматику у пам'ять. Представлення у пам'яті – у вигляді асоціативного масиву з елементами вигляду

нетермінал, правила

Для реалізації використати контейнер вигляду *map<char, string>*. Ключем у асоціативному масиві є нетермінал. З утвореного представлення утворити новий контейнер вигляду *vector<<para<char, string> >*. У рядок входить тільки одне правило (немає символів "|", а нетермінали у масиві можуть повторюватись). Вивести перетворену граматику у список TListBox. Перевірити, чи дійсно граматика праволінійна.

4. У багаторядковому полі редагування TMemo задано ліволінійну граматику. Кожному нетерміналу граматики відповідає один рядок TMemo. Правила у рядку розділяються символом "|". Зчитати граматику у пам'ять. Представлення у пам'яті – у вигляді асоціативного масиву з елементами вигляду

нетермінал, правило1, правило2,…

Для реалізації використати контейнер вигляду *map<char, vector<string> >*. Ключем у асоціативному масиві є нетермінал. Вивести множини нетерміналів і терміналів (контейнер *set<char>*). Згенерувати допустимий ланцюжок граматики, випадковим чином вибираючи правила для нетерміналів. Перевірити, чи дійсно граматика ліволінійна.

5. У багаторядковому полі редагування TMemo задано неорієнтований граф у вигляді інцидентних вершин

вершина 1 список інцидентних вершин
вершина 2 список інцидентних вершин
...
вершина n список інцидентних вершин

Зчитати його у пам'ять. Представлення у пам'яті – у вигляді масиву списків інцидентних вершин (контейнер *vector<list<int> >*). Ввести номери двох вершин графу і вивести шлях, по якому від першої вершини можна попасти у другу.

6. Неорієнтований граф заданий у вигляді списків інцидентних вершин (див. варіант 5). Зчитати його у пам'ять. Представлення в пам'яті – у вигляді масиву черг інцидентних вершин (контейнер *vector<queue<int> >*). Задано номер вершини. Увести додаткову нумерацію вершин: заданій вершині присвоїти номер 0, всім інцидентним з нею вершинам (шар 1) – номер 1, всім ще не переглянутим вершинам, інцидентним вершинам з номером 1 (шар 2) – номер 2 і т.д. Для цього зберігати і демонструвати на кожному кроці список вершин поточного шару. Вивести вершини по шарах.

7. У багаторядковому полі редагування TMemo записані слова через пробіл. Є деяка множина ключових слів, збережених в полі редагування TEdit. Створити таблицю зустрічальності для неключових слів у вигляді

слово (рядок, номер слова у рядку), ... , (рядок, номер слова у рядку)

і зберегти її у другому TMemo. Для збереження конструкцій виду *(рядок, номер слова у рядку)* використати контейнер *vector<pair<string,int> >*.

8. Розробити структуру даних «словник, поповнюваний словами з файлів». У словнику зберігати слово й кількість його повторень. Словник (контейнер *vector<pair<string,int> >*) повинен уміти себе зберігати у файл, відновлювати себе з файлу у багаторядкове поле редагування TMemo й видавати у другому TMemo або слова, упорядковані за алфавітом, або слова, упорядковані за по зустрічальності. Заголовок мітки, який позначає результат, змінювати в залежності від радіокнопки.

9. У багаторядковому полі редагування TMemo записані слова. Є таблиця синонімів, записана в іншому TMemo (для одного слова може бути кілька синонімів). Розробити структуру даних, що підтримує таблицю синонімів. Замінити кожне слово у першому TMemo на випадковий синонім і результат записати в третє TMemo. Використати зручний для використання контейнер бібліотеки STL, можливо такий – *map<string, vector <string> >*.

10. Є список студентів, що вивчають математику, фізику, біологію й хімію, заданий у файлі у вигляді рядків

прізвище, предмет

Зчитати файл у пам'ять. Представлення в пам'яті – контейнер *vector<pair<string, string> >*. Кожний предмет вивчає деяка кількість студентів. Розробити структури даних, що дозволяють відповідати на запити виду «Видати прізвища всіх студентів, що вивчають математику й фізику, але не вивчають програмування». Питання формуються за допомогою елементів керування типу TCheckBox, а результати виводяться у багаторядковому полі редагування TMemo.

11. Із заданої послідовності натуральних чисел утворити послідовність ключів (натуральних чисел, які не повторюються). Побудувати хеш-таблицю зі списками (використати контейнер *vector<stack<int> >*). Розмірність хеш-таблиці вводиться, а хеш-функція вибирається радіокнопками. У текстову таблицю TStringGrid вивести утворену хеш-таблицю.

12. Ввести координати точок на площині. Впорядкувати точки в залежності від радіокнопок або за відстанню від початку координат, або за зростанням кута між векторами з заданими координатами та віссю абсцис (використати контейнер *vector<pair<double, pair<int,int> >*). При необхідності (використати TCheckBox) знайти

відстані і/або кути. Області виводу, які не використовуються, зробити невидимими.

13. Із заданого тексту (контейнер *vector<string>*), який складається з окремих рядків, утворити текст з такими ж рядками, в яких залишити або всі голосні літери, або всі цифри, або всі символи без знаків пунктуації, які були у рядках (використати радіокнопки). Заголовок мітки, який позначає результат, змінювати в залежності від радіокнопки.

14. Ввести два масиви (контейнер *vector<float>*). Упорядкувати їх за спаданням або зростанням у залежності від ознаки (використати радіокнопки). При необхідності (використати ознаку) – злити в один упорядкований масив і знищити повтори елементів.

15. Дано масив (контейнер *vector<int>*) і номер позиції, з якої його треба перетворити. Впорядкувати масив у залежності від ознаки (використати радіокнопки) або в порядку зростання, або в порядку спадання, або реверсувати масив. Перенести перетворену частину на початок масиву.

16. Ввести натуральне число *n*. Вивести в різні списки (контейнер *list<int>*) взаємно прості з числом *n* числа, які менші за *n*; прості числа, які менші за *n*; додатні дільники числа *n*. Виводити результати, якщо включено відповідну ознаку. При вводі нового числа *n* очистити списки.

17. Перетворити введений масив чисел (контейнер *vector<int>*), зсунувши його циклічно на *k* позицій (*k* вводиться за допомогою форми DELPHI) вліво або вправо (в залежності від радіокнопки). Вивести перетворений масив в список.

18. У заданому тексті підрахувати і при необхідності вивести слова, які починаються і закінчуються однією буквою; містять тільки три букви; є симетричними. Знайдені групи слів зберігати у контейнерах типу *vector<string>*. Упорядковані групи слів виводити в три різні області в залежності від заданої ознаки.

19. Дано натуральне число *n*, дійсні числа $y_1,...,y_n$ (використати контейнер *vector<float>*). Обчислити одну з величин:

- суму всіх елементів, що належать відрізку [1,5];
- кількість від'ємних елементів;
- кількість елементів, що не належать відрізку [3,8].

Потрібний результат отримати при виборі відповідної радіокнопки. Використати одну область для виведення результату. Назву області змінювати.

20. Дано цілі числа $a_1,...,a_n$ (використати контейнер *vector<int>*). Одержати:

- кількість непарних і від'ємних;
- кількість елементів, що задовольняють умові $|a_i| < i^2$;
- суму чисел, кратних 5.

Потрібний результат отримати при виборі відповідного елемента типу TCheckBox. Утворити 3 області для виведення результатів. Області результатів, які не використовуються, зробити невидимими.

21. Дано рядок символів. Вивести або всі слова, в яких є однакові букви, або кількість слів, в яких усі букви різні (використати радіокнопки). Для цього застосувати контейнери *vector<string> і set<char>*. Назву вікна результатів міняти в залежності від інформації. Вивести повідомлення, якщо потрібних слів немає (використати форму DELPHI).

22. Дано рядок з слів і окреме слово. Вивести або всі слова, в яких є всі букви цього слова, або кількість слів, в яких немає жодної букви введеного слова (використати радіокнопки). Для цього застосувати контейнери *vector<string> і set<char>*. Назву вікна результатів міняти в залежності від інформації. Вивести повідомлення, якщо потрібних слів немає (використати форму DELPHI).

23. У полях редагування TEdit ввести три множини X1, X2, X3, що містять цілі числа (використати контейнер *set<int>*). Відомо, що потужність кожної із цих множин більша 10. Сформувати нову множину Y=(X1∪X2) (X2\X3), з якої виділити підмножину непарних чисел. Вивести вихідні й отримані множини, а також проміжні результати, якщо включено відповідну ознаку TCheckBox.

24. У полях редагування TEdit ввести три множини X1, X2, X3, що містять цілі числа (використати контейнер *set<int>*). Відомо, що потужність кожної із цих множин більша 10. Сформувати нову множину Y=(X1∪X2)\(X2·X3) і вивести її потужність. Побудувати множину Z з чисел, які в множині Y і діляться на 6 без остачі. Вивести вихідні й отримані множини, а також проміжні результати, якщо включено відповідну ознаку TCheckBox.

25. Задано послідовність одно-, дво- і тризначних натуральних чисел у полі редагування TEdit (використати контейнер *vector<int>*). Надрукувати множину цифр (використати контейнер *set<int>*), яка відповідає на запиту виду «Видати всі цифри, що входять у запис тризначних чисел і не входять у запис двозначних». Питання формуються за допомогою елементів керування типу TCheckBox.

26. Задано послідовність натуральних чисел у полі редагування TEdit (використати контейнер *vector<int>*). Сформувати множини цифр (використати контейнер *set <char>*)

- що входять у запис кожного числа;
- що входять у запис парних чисел і не входять запис непарних;
- що входять у запис хоча б одного числа більше одного разу.

Потрібний результат отримати при виборі відповідного елемента типу TCheckBox.

27. Задано непорожню послідовність слів з малих латинських літер. Між сусідніми словами – пробіли. Сформувати масив слів (контейнер *vector<string>*) і вивести його у список TList. Також вивести в алфавітному порядку множини літер (використати контейнер *set<char>*)

- букви, що входять до найдовших слів;
- голосні букви, що входять не більше, ніж до одного слова;
- букви, що входять до слів, які закінчуються голосною літерою.

Потрібний результат отримати при виборі відповідної радіокнопки. Використати одну область для виведення результату. Назву області змінювати.

28. Задано непорожню послідовність слів з малих латинських літер. Між сусідніми словами – кома. Сформувати масив упорядкованих слів (контейнер *vector<string>*) і вивести його у TMemo. Також вивести в алфавітному порядку множини літер (використати контейнер *set <char>*)

- дзвінкі приголосні букви, що входять тільки до одного слова;
- приголосні букви, що не входять до жодного слова;
- приголосні букви, що входять до слів, які починаються дзвінкою приголосною.

Потрібний результат отримати при виборі відповідного елемента типу TCheckBox.

29. Задано непорожню послідовність слів з малих латинських літер. Між сусідніми словами – знаки пунктуації. Сформувати масив слів (контейнер *vector<string>*) і вивести його у список TList. Також вивести в алфавітному порядку множини слів (використати контейнер *set <string>*)

- що мають парну кількість букв;
- які є симетричними;
- що складаються з різних символів.

Потрібний результат отримати при виборі відповідної радіокнопки. Використати одну область для виведення результату. Назву області змінювати.

30. Задано непорожню послідовність слів. Кожне слово – послідовність цифр. Між сусідніми словами – знаки пунктуації. Вивести множини слів (використати контейнер *set <string>*)

- що є щасливими (мають однакову суму цифр в кожній з половин слова);
- які складаються з цифр, що є простими числами;
- які складаються з парних цифр.

Сформовані множини слів вивести у списках TList.

Потрібний результат отримати при виборі відповідного елемента типу TCheckBox.

Лабораторна робота №2

Тема: Використання динамічно створених компонентів

Побудувати Windows-додаток з багатодокументним інтерфейсом (MDI). Для цього

- перетворити звичайну форму (FormStyle = fsNormal) з лабораторної роботи №1 у дочірню форму (FormStyle = fsMDIChild);
- створити батьківську форму (FormStyle = fsMDIForm) з декількома пунктами меню (наприклад, File, Windows, About), в тому числі з пунктом About, за допомогою якого викликається форма з умовою задачі;
- зробити батьківську форму головною формою проекту;
- на дочірній формі використати динамічно створені компоненти і динамічне переключення обробників подій (для цього при необхідності доповнити умову задачі);
- підключити до проекту форму, побудовану в середовищі Delphi, в якій розмістити незначну частину розробки (наприклад – умову задачі);
- у різних екземплярах дочірньої форми, які динамічно створюються при виборі відповідних елементів підменю Windows, підготувати тестові приклади для демонстрації роботи програми.

Лабораторна робота №3 (для магістрів)

Тема: Створення компонентів користувача

Створити компонент користувача із вказаними властивостями методами і подіями (інших подій, крім заданих, компонент не повинен мати). В якості пращура краще брати схожий абстрактний клас. Протестувати створений компонент у власному проекті до інсталяції його на палітру. Інсталювати компонент на сторінку Samples палітри компонентів. Випробувати інстальований компонент.

Варіанти завдань

1. Компонент – заокруглений прямокутник з надписом. Властивості компонента – розміри прямокутника, текст надпису, колір фону, колір тексту. Методи – зміна тексту, обертання тексту. Подія – OnClick.
2. Компонент – кільце. Властивості компонента – зовнішній радіус, ширина кільця, колір фону, колір кільця. Методи – зміна кольору кільця, зміна ширини кільця. Подія – OnClick.
3. Компонент – рівносторонній трикутник. Властивості компонента – довжина сторони, колір заповнення, стиль заповнення. Методи – поворот, зміна довжини сторони, зміна кольору заповнення Подія – OnClick.
4. Компонент – прямокутник. Властивості компонента – довжини двох сторін, колір заповнення, стиль заповнення, колір контуру. Методи – обмін сторін (swap), зміна довжин сторін, зміна кольору заповнення і кольору контуру. Подія – OnClick.
5. Компонент – еліпс. Властивості компонента – довжини півосей, колір заповнення, стиль заповнення. Методи – обмін півосей (swap), зміна довжин півосей, зміна стилю заповнення і кольору контуру. Подія – OnClick.
6. Компонент – сектор круга. Властивості компонента – радіус, початковий кут, кінцевий кут, колір заповнення, стиль заповнення. Методи – поворот, зміна кінцевого кута, зміна кольору заповнення Подія – OnClick.
7. Компонент – сегмент круга. Властивості компонента – радіус, центральний кут, початковий кут, колір заповнення, стиль заповнення. Методи – поворот, зміна центрального кута. Подія – OnClick.
8. Компонент – спеціалізоване поле редагування і вводу TEdit для масиву цілих чисел. Властивості компонента – Text з цифрами і пробілами, розмірність, масив утворених цілих чисел, максимум масиву. Методи – сортування масиву, реверсування масиву. Подія – OnChange.
9. Компонент – спеціалізоване поле редагування і вводу TEdit для дійсних масиву чисел з дробовою крапкою. Властивості компонента – Text з послідовності дійсних чисел у символьному вигляді, розмірність, масив утворених дійсних чисел, мінімум цілої частини елементів масиву. Методи – формування рядка з

цілими частинами, перевірки відсутності у чисел дробової частин. Подія – OnChange.

10. Компонент – годинник з часовою і хвилинною стрілками Властивості компонента – радіус, кути часової і хвилинною стрілок. Методи – встановлення поточного часу, зміна часу на задану кількість хвилин. Подія – OnClick.

11. Компонент – електронний годинник з показаннями годин, хвилин і секунд. Властивості компонента – формат часу для демонстрації (два типи формату), години, хвилини, секунди. Методи – встановлення поточного часу, зміна часу на задану кількість хвилин, зміна формату часу. Подія – OnClick.

12. Компонент RegExpMemo – багаторядкове поле редагування з можливістю використання пошуку за шаблоном з метасимволами. Властивості компонента – рядок з шаблоном, індекс початку для пошуку ланцюжка, колір виділення. Методи – пошук ланцюжка, що відповідає шаблону, довжина знайденого ланцюжка, зміна індексу початку пошуку. Подія – OnChange. Для реалізації задачі використати клас regexp, підключивши його до проекту директивою #include <regexp>.

Лабораторна робота №3 (для спеціалістів)

Тема: Використання графіки

Варіанти завдань

13. Зобразити перетворення квадрата в круг. Радіус круга і швидкість перетворення задається користувачем.

14. Зобразити вільне падіння краплі на тверду поверхню. Масу краплі та її початкову швидкість задає користувач.

15. Зобразити рух кульки по еліптичній траєкторії. Надати користувачу можливість задавати швидкість кульки та ексцентриситет еліпсу.

16. Зобразити обертання колеса навколо своєї осі. Радіус колеса та швидкість обертання задаються користувачем.

17. Намалювати хвилю (синусоїду) яка рухається. Напрямок, швидкість руху та висоту хвилі має вибирати користувач.

18. Зобразити спіраль Архімеда, що обертається навколо свого центру. Напрям, швидкість обертання, а також параметри спіралі задаються користувачем.

19. Зобразити рух маятника із врахуванням сили тяжіння Землі. Маса, довжина, початкове положення і швидкість маятника задаються користувачем.

20. Зобразити зірку яка обертається навколо своєї осі симетрії. Надати можливість користувачу визначати швидкість обертання та розміри зірки.

21. Зобразити олівець якій малює синусоїду $a \cdot sin(b \cdot x)$. Надати користувачу можливість задавати значення параметрів *a* та *b*.

22. Зобразити рух кульки всередині поверхні, обмеженої колом. Радіус кола, початкова швидкість кульки, напрям, її початкове положення задаються користувачем.

23. Зобразити збільшувальне скло (коло) яке рухається над рядком тексту. Літери, які потрапляють в коло, зображати вдвічі більшими. Рядок тексту та швидкість руху кола задає користувач.

24. Зобразити куб якій обертається навколо осі OZ. Надати користувачу можливість задавати довжину ребра куба та швидкість обертання.

25. Зобразити процес накачування м‘ячика. Критичний тиск у м‘ячику і швидкість накачування задаються користувачем.

26. Зобразити рух декількох кульок всередині області, обмеженої прямокутником. Початковий напрямок руху кульок вибирається випадковим чином. Розміри прямокутника, швидкість руху кульок, а також їх кількість (від 1 до 5) задається користувачем.

27. Зобразити перетворення круга в зірку зі збереженням площі. Радіус круга і швидкість перетворення задаються користувачем.

28. Зобразити процес вимальовування графіків деяких функції функцій на одній координатній площині різними кольорами. Для вибору функцій використати TCheckBox. Для вибору кольорів застосувати вбудоване стандартне діалогове вікно.

29. Зобразити відкриття/закриття парасолі (по натисканню клавіші Enter) і поворот парасолі праворуч/ліворуч (по натисканню правої/лівої клавіш миші).

30. Розробити простий графічний редактор, який дозволяє компонувати малюнки за допомогою динамічно створених компонентів типу TShape. Дати можливість вибирати тип кожної фігури (властивість Shape), з яких складається майбутній малюнок, та її колір (групи радіокнопок).

Лабораторна робота №4

Тема: Доступ до баз даних

Розробити базу даних у довільному середовищі (Access, Visual FoxPro і т.і.). Занести у таблицю бази даних не менше 10 записів. Для доступу до бази даних встановити псевдонім (alias) через адміністратор джерел даних ODBC. У проекті C++Builder до створеної бази даних звертатися за її псевдонімом, використовуючи механізми доступу із застосуванням ADO + OLE DB + ODBC для непарних номерів і BDE + ODBC Link + ODBC для парних номерів.

У проекті

- створити дві форми: відокремити модуль даних та інтерфейсну частину проекту;
- створити SQL-запити, використовуючи властивість елемента керування типу TQuery або елемента керування типу TADOQuery;
- використати додаткове обчислювальне поле (Calculate) в таблицях баз даних, добавлене з середовища C++Builder;
- використати фільтри (властивості Filter і Filtered);
- створити динамічні запити з параметрами (властивість Params для компонента TTable, або властивість Parameters для компонента TADOTable).

Для реалізації вище перерахованого при необхідності доповнити умову задачі.

Варіанти завдань

1. Відомості про книги – це прізвище автора, назва, рік видання, вартість. Є база даних з таблицею, записи якої – відомості про книги. Створити запит на книги автора Іванова 2006-2009 років видання, впорядкувавши назви за алфавітом. Надрукувати інформацію про всі книги, знизивши їх вартість на 10% (обчислювальне поле).

2. Інформація про автомобіль складається з номера, марки, року випуску прізвища. Є база даних з таблицею, записи якої – інформація про автомобілі. Створити запит на автомобілі ВАЗ з номерами, більшими 8800. Надрукувати інформацію про всі автомобілі у хронологічному порядку років випуску. Добавити

обчислювальне поле з номерами, в яких збільшити всі номера (без буквеної частини) на 100.

3. Відомості про учня складаються з його прізвища , імені, домашньої адреси, номера телефону. Є база даних з таблицею, записи якої – інформація про учнів. Створити запит на прізвища учнів з вулиці Головна, упорядкувавши їх за алфавітом. Створити нове обчислювальне поле в таблиці, об'єднавши адресу з телефоном. Вивести таблицю з прізвищем і новим полем.

4. У таблиці задано інформацію про назву міста, кількість інститутів, чисельність населення, чисельність студентів. Впорядкувати таблицю по спаданню кількості інститутів, створивши в ній нове обчислювальне поле: відношення кількості студентів до чисельності населення. Створити запит на назви міст з кількістю інститутів більшою за 3.

5. Є база даних з таблицею, записи якої – інформація про ліки, які зберігаються на складі: назва, термін зберігання, дата випуску, кількість одиниць, вартість. Створити запит на ліки, для яких вже вийшов термін зберігання. Вивести таблицю з назвами ліків та їх загальною вартістю (створити обчислювальне поле).

6. На складі зберігається продукція заводу. Є база даних з таблицею, записи якої – інформація про кожний вид продукції: номер продукції, назва, кількість одиниць, дата випуску, вартість одиниці. Створити запит на назви продукції, випущеної з 2007 по 2009 роки, відсортувавши їх за алфавітом. Створити нове обчислювальне поле з повною вартістю продукції. Вивести таблицю з номером продукції та новим полем.

7. Є база даних з таблицею, записи якої – інформація про точки у просторі: координати точки, назва точки, маса точки. Створити запит на записи назвами і масами точок, впорядкований по назвах. Вивести таблицю з назвами точок і відповідними відстанями від точок до початку координат (створити обчислювальне поле).

8. Є база даних з таблицею, записи якої – інформація про прямі у просторі: коефіцієнти рівняння прямої, назва прямої, колір прямої. Створити запит на записи з назвами і кольором прямих, які належать координатним площинам. Впорядкувати записи по назвах. Вивести таблицю з назвами прямих і відповідними відстанями від прямих до початку координат (створити обчислювальне поле).

9. Є база даних з таблицею, записи якої – інформація про файли: назва файлу, розширення, дата створення, розмір, атрибут. Створити запит на файли з розширенням .exe, розмір яких більший за 100 Кб. Впорядкувати записи по назвах. Вивести таблицю з назвами файлів і примітками (створити обчислювальне поле) про то, чи файл можна виконувати , чи ні (розширення .exe, .bat, .com).

10. Є телефонна база даних з таблицею, записи якої складаються з п'ятизначного номера телефона, прізвища або назви організації, адреси. Створити запит на список номерів, які починаються на 3, впорядкувавши прізвища за алфавітом. Вивести таблицю, змінивши всі номери, які починаються на 2 на такі ж, але з першою цифрою 5. Завести для цього додаткове обчислювальне поле.

11. Є база даних електромережі з таблицею, записи якої складаються з прізвища квартиронаймача, початкових і кінцевих показників лічильника за місяць, тарифу, кількості кіловат. Створити запит на прізвища квартиронаймачів, у яких кінцевий показник лічильника більший за 12345, впорядкувавши їх за алфавітом. Вивести таблицю з новим полем, в якому визначається сума до оплати (створити обчислювальне поле).

12. Є база даних по навантаженню викладача з таблицею, записи якої складаються з назви предмету, номера курсу, кількості студентів, кількості лекційних і лабораторних годин, ознаки наявності заліку або іспиту. Створити запит на предмети, які викладаються на 5 курсі і для яких є залік, впорядкувавши назви за алфавітом. Вивести таблицю, зменшивши кількість лабораторних годин на 4 (створити обчислювальне поле).

13. База даних містить таблицю з інформацією про кімнати у гуртожитку. Запис таблиці має вигляд: кількість мешканців, максимально можлива кількість мешканців, номер кімнати, площа кімнати, факультет. Створити запит про кімнати, у яких найменша площа на одного студента, згрупувавши записи за факультетами. Вивести таблицю, добавивши обчислювальне поле з кількістю незайнятих місць у кімнаті.

14. База даних містить таблицю з інформацією про квартири у домі. Запис має вигляд: прізвище власника, кількість зареєстрованих мешканців, площа, число кімнат, поверх. Вивести дані про квартири із площею менше 50 кв. м., впорядкувавши записи за кількістю кімнат. Вивести таблицю, додавши в неї обчислювальне поле з площею, яка припадає на кожного мешканця квартири.

15. База даних містить таблицю з інформацією про поїзди. Запис має вигляд: номер потягу, пункт призначення, час відправлення, час прибуття, кількість вільних місць. Отримати довідку про наявність вільних місць на заданий номер потягу. Вивести дані про потяги із кількість вільних місць більшої 60, впорядкувавши записи за часом відправлення. Вивести таблицю, добавивши в неї обчислювальне поле з часом у дорозі.

16. База даних – відомості про птахів. У таблиці міститься інформація про назву птаха, масу, розмах крила, ціну. Вивести інформацію про всіх птахів; інформацію про птахів, розмах крил яких більше заданого. Створити обчислювальне поле – відношення маси до розмаху крила.

17. База даних – інформація про кактуси в оранжереї. У таблиці містяться поля: назва кактуса, час посадки, висота, ціна, колір, колючість, кількість. Вивести інформацію про всі кактуси. Вибрати кактуси старші 5 років. Створити обчислювальне поле – вік кактуса.

18. База даних – інформація про ботанічний сад. У таблиці містяться поля: назва дерева, час посадки, висота, товщина стовбура. Вивести інформацію про дерева. Вибрати дерева з товщиною стовбура більшою за 15см. Додати обчислювальне поле – вік дерева.

19. База даних – інформація про телевізори. У таблиці містяться поля: марка, фірма-виробник, дата випуску, гарантійний термін, ціна, кількість одиниць на складі. Вибрати телевізори з ціною більшою за 100$. Додати обчислювальне поле – час, який залишився до кінця гарантійного терміну.

20. База даних – інформація про канцелярські приладдя. У таблиці містяться поля: назва, кількість, вартість, виробник, постачальник. Вибрати канцелярські приладдя з ціною більшою за 10 грн. Додати обчислювальне поле – вартість товарів певної назви.

21. Є база даних спортивних шкіл (номер, адреса, кількість учнів, рік заснування). Створити запит „кількість учнів у спортивних школах" (номер школи, кількість учнів), впорядкувавши кількість учнів за спаданням. Відобразити п'ять найстаріших шкіл.

22. Є база даних „Вчитель". Таблиця містить інформацію про учнів: порядковий номер, ПІП, оцінки за предмети (математика, фізика, хімія, біологія, історія). Створити запит, що виведе інформацію про успішність учнів по математиці (ПІП, оцінка, впорядкована за спаданням). Засобами C++Builder створити обчислювальне поле – середня оцінка (№, ПІП, середня оцінка).

23. Є база даних „Нерухомість". Таблиця містить інформацію про доступну нерухомість: тип, адреса, площа, ціна. Створити запит – 5 найдорожчих пропозицій. Засобами C++Builder створити обчислювальне поле – середня ціна 1 кв.м. (№, ПІП, середня оцінка).

24. Є база даних „Комп'ютери". Таблиця містить інформацію: назва, тип, кількість процесорів, об'єм оперативної пам'яті, об'єм вінчестера, наявність CD-ROM, рік випуску, ціна. Створити запит, що виведе інформацію про вік комп'ютерів. Засобами C++Builder створити обчислювальне поле, що характеризує співвідношення об'єму оперативної пам'яті до об'єму вінчестера .

25. Є база даних „Користувачі ПК". Таблиця містить інформацію: логін, пароль, ПІП користувача, дата реєстрації, дата останнього

входу в систему. Вивести інформацію про 5 найактивніших користувачів.

26. Є база даних „Книги”. Таблиця містить інформацію: назва, автор, рік видання, дата поступлення в магазин, ціна за одиницю, кількість на складі. Створити запит, що виведе інформацію про 10 найстаріших книг (за роком видання). Засобами C++Builder створити обчислювальне поле, що характеризує сумарну вартість книг для кожної назви.

27. Є база даних „Веб-сайти”. Таблиця містить інформацію: назва сайту, веб-адреса, кількість відвідувачів у день. Створити запит, що виведе інформацію про 10 найпопулярніших сайтів. Добавити обчислювальне поле ”кількість символів у назві сайту”.

28. Є база даних „Картини”. Таблиця містить інформацію: назва картини, дата написання, автор, дата поступлення, вартість. Вивести 10 найстаріших картин і їхню вартість. Добавити обчислювальне поле ”час знаходження у магазині”.

29. Є база даних „Туристичні маршрути”. Таблиця містить інформацію: назва маршруту, кількість місць, вартість, кількість днів, наявність знижок для постійних клієнтів, можливість сімейного відпочинку. Створити запит на туристичні маршрути із знижками без можливості сімейного відпочинку. Добавити обчислювальне поле ”вартість одного дня у маршруті”.

30. Є база даних „Транспортні маршрути”. Таблиця містить інформацію: маршрут, початкова станція, кінцева станція, час відправлення, час прибуття, вартість квитка. Добавити обчислювальне поле ”час у дорозі”. Вивести 5 найтриваліших маршрутів та їхню вартість.

Список літератури

1. Архангельский А.Я. Программирование в C++Builder 6.– М.: ЗАО ”Издательство БИНОМ”, 2002.–1152с.
2. Архангельский А.Я. C++ Builder 6. Справочное пособие. Книга 1. Язык C++. – М.: Бином-Пресс, 2004 г.– 544 с.
3. Мєтт Теллес. Borland C++Builder: библиотека программиста. – СПб: ПитерКом,1998. – 512с.
4. Кент Рейсдорф, Кен Хендерсон. Borland C++Builder: Освой самостоятельно. – Москва: Бином, 1998.–702с.
5. Буч Г. Объектно-ориентированное проектирование с примерами применения.– М.: Конкорд, 1992. – 367 с.
6. Шамис В.А. C++Bulder 4. Техника визуального программирования. Издание второе, переработанное и дополненное.– М.: Нолидж, 2000. – 656 с.
7. Шамис В. C++ Builder Borland Developer Studio 2006. – СПб: Издательство "Питер", 2007. – 784 с.
8. Алексанкин В. Г., Елманова Н. З. Среда разработки C++ Builder. – СПб: Издательство "Питер", 1999. – 312 с.
9. Тимофеев В.В. Язык C/C++. Программирование в C++Builder 5. – Москва: Бином, 2000. – 368 с.
10. Кошель С.П., Елманова Н.З. Введение в Borland C++ Builder. – М.: Диалог-МИФИ, 1997.– 252 с.
11. Страуструп Б. Язык программирования C++. – К.: ДиаСофт, 1993. – 256 с.
12. Страуструп Б. Язык программирования C++. – СПб.: ”Невский Диалект”, 2002.– 1099 с.
13. Страуструп Б. Дизайн и эволюция C++. – СПб: Издательство "Питер", 2006.– 448с.
14. Скотт Мейерс. Эффективное использование STL. Библиотека программиста . – СПб: Издательство "Питер", 2003. – 400 с.
15. Мэтью Г. Остерн. Обобщенное программирование и STL. Использование и наращивание стандартной библиотеки шаблонов C++. – СПб.: Невский Диалект, 2004. – 544 с.

Інтернет-ресурси

16. http://solarix.ru/for_developers/cpp/stl/stl.shtml – Руководство по стандартной библиотеке шаблонов (STL)
17. http://www.math.rsu.ru/mexmat/miks/CPP/– С.С. Михалкович. Стандартная библиотека C++
18. http://users.msu.dubna.ru/~ovg/c++course – В. Ольшевский. Курс C++
19. http://www.realcoding.net/article/view/1844 – Итераторы библиотеки STL
20. http://informatics.mccme.ru/moodle/mod/book/view.php?id=492&chapterid=200 – Д. Королёв. Использование библиотеки STL
21. http://www.mywebsite.org.ua/article/vykorystannya-stl-v-c_plus_plus/index.html – Використання STL в C++
22. http://mf.grsu.by/other/lib/db – А. Федоров, Н.Елманова. Введение в базы данных
23. http://www.research.att.com/~bs/papers.html – Публикации Бьерна Страуструпа
24. http://www.kv.by/index2000031102.htm – С.Соколов. Визуальное программирование
25. http://www.williamspublishing.com/PDF/5-8459-0499-4/part.pdf – Основы C++Builder
26. http://ruseti.ru/book1/ – C/C++ Программирование в среде C++ Builder 5
27. http://www.piter.com/lib/978531400107/ – М. Теллес. Borland C++ Builder. Библиотека программиста
28. http://www.ru-coding.com/c_1.php – Программирование / C Builder учебник
29. http://www.mywebsite.org.ua/catalog/cvccsharp-pidruchnyk-po-c_plus_plus-builder/ – Підручник по C++ Builder
30. http://window.edu.ru/window_catalog/files/r47642/pstu004.pdf – А.М. Ноткин. Объектно-ориентированное программирование на языке C++
31. http://c-books.info/books/load.php?ty=c&lng=RU – Электронные книги: C-RU
32. http://fti.org.ua/pub/1%20kurs/%cf%f0%ee%e3%f0%e0%ec%ec%e8%f0%ee%e2%e0%ed%e8%e5/%ce%ce%cf%20%ed%e0%20%d1++/%ca%ed%e8%e3%e8%20%ef%ee%20%d1++/ – Книги по C++

Навчальне видання

Сопронюк Тетяна Миколаївна

Технології візуального й узагальненого програмування в C++Builder

Навчальний посібник

Відповідальний за випуск*: Бігун Я. Й.*

Комп'ютерний набір*: Сопронюк Т. М.*

Підписано до друку .01.2009. Формат 60 х 80/16
Папір газетний. Друк офсетний. Ум. друк. арк. 5. Обл.-вид. арк. 5.
Зам.131. Тираж 100. Безплатно.
Друкарня видавництва "Рута" Чернівецького національного університету
58012, Чернівці, вул.Коцюбинського, 2