В. П. Семеренко

ПРОГРАМУВАННЯ МОВАМИ

С та С++

В СЕРЕДОВИЩІ WINDOWS

Міністерство освіти і науки України
Вінницький державний технічний університет

В. П. Семеренко

ПРОГРАМУВАННЯ МОВАМИ

C та C++

В СЕРЕДОВИЩІ WINDOWS

Затверджено Ученою радою Вінницького державного технічного університету як навчальний посібник для студентів бакалаврського напрямку 6.0915 - "Комп'ютерна інженерія" всіх спеціальностей. Протокол № 5 від 27 грудня 2001 р.

Вінниця ВДТУ 2002

УДК 681.3.06.(075)
С30

Рецензенти:

С.В. Юхимчук, доктор технічних наук, професор
А. М.Пєтух. доктор технічних наук, професор
Р.Г. Тадевосян, кандидат фізико-математичних наук, доцент

Рекомендовано до видання Ученою радою Вінницького державного технічного університету Міністерства освіти і науки України

Семеренко В.П.

С30 **Програмування мовами С та С++ в середовищі Windows**.- Навчальний посібник. - Вінниця: ВДТУ, 2002.- 128 с.

В посібнику розглянута методика складання програм мовою С з використанням функцій API Windows та мовою С++ з використанням бібліотеки класів MFC програмного пакета Visual C++ 6.0. Даються основи об'єктно-орієнтованого програмування мовою С++. Посібник призначений для студентів спеціальності 7.0911501 "Комп'ютерні системи та мережі" для вивчення дисциплін "Системне програмне забезпечення" і "Програмування", а також може бути рекомендований студентам інших спеціальностей, пов'язаних з вивченням сучасного програмного забезпечення.

УДК 681.3.06.(075)

ЗМІСТ

Передмова

Писати підручники посібники по сучасному програмуванню ризиковано і важко. Ризиковано, тому що вік таких книг, як правило, недовгий. Нові мови програмування, нові стилі програмування, нові програмні пакети змінюються з калейдоскопічною швидкістю. Щоб нові книги по програмному забезпеченню були корисними, вони повинні видаватись та оновлюватись такими ж швидкими темпами.

Чисельні видавництва випускають велику кількість книг і на перший погляд здається, що важко знайти ті сфери, де існує гостра потреба в навчальній літературі.

Однак при прискіпливому аналізі можна помітити деякі загальні закономірності.

Вимога ринкових відносин бути першими в представленні чергового програмного продукту змушує деяких авторів використовувати "конвейерні" способи написання нових книг. Цей спосіб полягає в сумлінному перерахуванні всіх існуючих режимів роботи нового програмного продукту, всіх форматів команд, всіх пунктів меню, всіх керуючих кнопок і т. д. По суті книга перетворюється в звичайний довідник, який мало відрізняється від документації фірми-виробника. Безумовно, такі книги теж потрібні, однак вони мало допоможуть у навчанні програмуванню, особливо на першому етапі.

Існують також книги, основний зміст яких складається з програмного коду з невеликими коментарями. Як правило, ці програми є копіями програм, що їх автоматично генерують різні "Майстри" (Wizard) програмних пакетів. Такий "Майстер" може швидко створити програмний код під невеликий набір стандартних завдань користувача. Зате в результаті утворюються великі за обсягом програми, в яких важко розібратись новачку, а тим більш внести свої корективи. Звичайно, аналіз таких програм теж корисний та цікавий.

І все ж для того, щоб навчитись самому програмувати і створювати ефективні програми, варто йти іншим шляхом.

Методика навчання програмування, подібно іншому предмету, передбачає рух від простого до складного. Спочатку необхідно спробувати самостійно написати найпростішу, але працюючу програму. Подальше засвоєння нових тем і розділів буде ускладнювати початкову просту програму. Самостійно змінюючи власну програму та додаючи до неї нові можливості, можна в кінцевому результаті навчитись створювати досить складні програми.

Неоціненну роль в такому процесі навчання і повинні відіграти підручники і навчальні посібники. І ось тут виявляється, що підготувати такі навчальні книги дуже непросто. Адже, з однієї сторони, необхідно мати відпрацьовану роками методику вивчення конкретної мови програмування, а з іншої сторони, швидке моральне старіння програмних

продуктів встановлює дуже жорсткі строки в підготовці навчальних методик і програмних текстів.

Найбільш корисні і вдалі книги з вивчення мов програмування, зокрема мов С та С++, пишуть ті автори, які постійно працюють в цій області програмного забезпечення. Тому кожна їх нова книга лише доповнює і збагачує новими матеріалом попередню. Це справедливо, зокрема, до відомого у світі програміста Герберта Шилдта. В його книгах є все: і доступність викладення, і детальний аналіз різних "підводних каменів" в програмах, і повні тексти програм, причому працюючих програм. І недаремно кожна книга цього чудового програміста стає справжнім бестселером.

Створюючи цей навчальний посібник, автор теж намагався писати в стилі Г. Шилдта. В якій мірі цього вдалося досягти, вирішувати читачам.

Методичні рекомендації до вивчення навчального посібника

Для успішного засвоєння матеріалу даного посібника необхідні базові знання з програмування та вміння складати програми мовою С у традиційному процедурному стилі. Читач має бути знайомим з програмуванням розгалужень і циклів, вміти працювати з функціями та файлами.

Навчальний посібник складається з чотирьох розділів, з яких другий і четвертий розділи є основними, а інші розділи – допоміжними.

Перший розділ буде корисний тим, хто вперше починає працювати з програмним пакетом Visual C++ 6.0. У цьому розділі основна увага приділена практичним порадам по створенню проектів для 32-розрядних програм Windows та виконанню програм в різних режимах налагодження.

Автор відмовився від того, щоб дати хоча б коротку характеристику всім командам головного меню, всім керуючим кнопкам і всім інструментам інтегрованого середовища. По-перше, це потребує значного обсягу матеріалу, по-друге, вся довідкова інформація вже наведена в чисельних книгах і довідниках з Visual C++ 6.0. Але найголовніший аргумент такої відмови полягає в тому, що для створення нескладних прикладних програм в в середовищі Windows зовсім не обов'язково спочатку вивчити всі команди інтегрованого середовища. Знайомство з новими командами буде відбуватися поступово, по потребі, можливо, що у використанні деяких команд чи режимів роботи взагалі не виникне потреба. Саме тому спочатку достатньо добре засвоїти послідовність створення лише кількох конкретних типів проектів.

Другий розділ посібника є основним, оскільки присвячений розробці програм мовою С в середовищі Windows. Тут детально пояснюється структура API-програм, принципові відмінності програмування в операційних системах Windows 95/98/ME від інших систем. Спочатку

увага звертається на програмування векторної графіки і тексту в API-програмах. Далі розглядається створення проектів з використанням найбільш поширених видів ресурсів: меню та діалогу.

Для того, щоб зрозуміти ідею створення програм з використанням бібліотеки класів MFC пакета Visual C++ 6.0, необхідно спочатку добре засвоїти принципи об'єктно-орієнтованого програмування (ООП). Цим питанням присвячений ще один допоміжний розділ посібника – третій розділ. Тут пояснюються три складові частини ООП: інкапсуляція, успадкування і поліморфізм. На відміну від першого допоміжного розділу, цей розділ можна вважати теоретичним. Приклади приведених тут програм не орієнтовані на роботу в середовищі Windows і можуть бути виконані в найпростішій однозадачній операційній системі, наприклад в MS-DOS. Якщо читачі вже знайомі з основами ООП, тоді третій розділ можна пропустити і приступити відразу до вивчення четвертого розділу.

Останній розділ посібника, який присвячений програмуванню мовою C++ з використанням бібліотеки класів MFC, є найскладнішим. Він потребує великої уваги і творчого підходу до вивчає мого матеріалу. Теми кількох підрозділів четвертого розділу (створення меню та діалогових вікон, програмування векторної графіки) схожі з однойменними темами в другому розділі. Це дає змогу порівняти різні підходи до виконання однакових задач, відчути переваги, що надає бібліотека класів MFC. В останньому розділі розглядається також і ряд нових тем.

Засвоїти програмування в операційних системах Windows 95/98/ME неможливо тільки теоретично, навіть при наявності всієї виданої документації. Вивчення програмування невіддільне від практики. Тому найкращий варіант роботи з даним посібником може бути тільки разом із комп'ютером. Перевіряючи наведені в посібнику фрагменти програм і змінюючи їх, можна зрозуміти принципи сучасного програмування.

В цілому даний посібник буде корисний при перших кроках в програмуванні мовами C та C++ в середовищі Windows. Для більш глибокого вивчення всіх тонкощів цього виду програмування необхідно, звичайно, користуватись і іншими книгами і посібниками, кращі з яких наведені в списку літератури.

Вступ

Головна задача сучасного програмування – це створення багатократно використовуваних незалежних елементів, придатних для складання різноманітних конкретних програм [1].

Традиційна технологія програмування зароджувалась в умовах, коли програма створювалась під одну конкретну задачу, а спосіб створення самої програми не мав принципового значення. Створення програм в ті далекі вже роки було майже мистецтвом, доступним вузькому колу професіоналів.

Однак в міру збільшення обсягу програм та необхідності модернізації раніше розробленого програмного забезпечення традиційний підхід до програмування швидко себе вичерпав. Колектив програмістів міг ефективно працювати над одним проектом лише тоді, коли всі члени колективу притримувались єдиної методики створення програм. Тільки в таких умовах програмні модулі окремих програмістів могли бути зістиковані разом за короткий проміжок часу і практично без помилок. Розробка єдиного методу на всіх етапах життєвого циклу програмного проекту – специфікації, проектування, власне програмування (кодування) і тестування – свідчила про появу нового стилю програмування, який отримав назву структурного програмування. З'явившись в 70-х роках двадцятого століття структурний підхід дозволив надати виробництву програм індустріальний характер.

Однак з часом все помітнішою ставала обмеженість структурного підходу, особливо коли постала необхідність повторного використання раніше розроблених програмних модулів. Ця потреба, а також і інші вади структурного програмування стимулювали появу нової технології – об'єктно-орієнтованого програмування (ООП). На відміну від бібліотек стандартних підпрограм, в яких теж використовуються повторні модулі, об'єктний підхід дозволяв створити ще ієрархію вкладених один в одного модулів.

Основи ООП були закладені ще на початку 80-х років, але і досі продовжуються дискусії з приводу того, чи повністю виправдала нова технологія покладені на неї великі надії. Опоненти ООП часто звертають увагу, зокрема, на складність програм, написаних в цьому стилі. І дійсно, саме намагання спростити процес написання програм було однією з вагомих причин появи на початку 90-х років нового напрямку в програмуванні – компонентного програмування (КП). В основі КП лежить досить цікава ідея – повторно використовувати можна не тільки програмний код модуля, але і готові результати модуля.

Таким результатом роботи модуля можуть бути, наприклад, елементи інтерфейсу програми: меню, командні кнопки, графічні зображення тощо. В такому разі програму вже не треба писати на папері, а досить лише її "побудувати" з готових "будівельних блоків" на екрані дисплея.

Більше того, компілятор ще й самостійно згенерує програмний код під створену таким чином програму. В цьому полягає суть візуального програмування – найбільш поширеного варіанта КП.

Звичайно, програмісту ще теж вистачить роботи, хоча процес підготовки програм вже значно спрощується. Зрозуміло, що таке спрощення програмування стосується в першу чергу інтерфейсних програм, в інших випадках застосування візуального програмування може бути недоцільним.

Існує чимало пояснень причин появи кожного стилю програмування. У короткому вступі неможливо провести аналіз всіх цих причин, навести приклади інших підходів у сучасному програмуванні. Підсумовуючи короткий історичний аналіз необхідно відмітити, що сьогодні існують поруч різні стилі і напрямки у програмуванні і кожна нова технологія не відміняє попередню. В багатьох сучасних програмних пакетах використовуються елементи різних технологій і їх вибір залежить в першу чергу від особливостей поставленої задачі. Наприклад, в пакеті Visual C++ 6.0, вивченню якого присвячений цей навчальний посібник, використовується і ООП, і КП.

Історія програмування пов'язана не тільки з розвитком стилів, але і з розробкою самих мов програмування. За останні 50 років було створено кілька тисяч мов програмування [2]. Світове визнання отримали лише кілка десятків мов програмування, які знайшли своє застосування на різних типах комп'ютерів.

Обмежимо наш подальший аналіз лише універсальними мовами програмування, тобто мовами високого рівня. В основу класифікації таких мов можна покласти одне з фундаментальних понять в теорії програмування – поняття типів даних.

За загально прийнятним визначенням, тип даних – це множина значень, які приймають дані, множина операцій над даними і розмір пам'яті, які займають дані. З точки зору можливості зміни даних програмістом, дані поділяються на декілька категорій.

До першої категорії відносять дані, які програміст не має права змінювати. Це так звані базові типи даних: цілі, дійсні, символьні, булеві. Деякі з них можуть утворювати однорідні (тобто із одного типу) об'єднання – масиви. Ті мови, які містять лише базові типи даних, називаються мовами програмування 1-го покоління. Найбільш відомі мови 1-го покоління – Фортран, Алгол-60, Бейсик. Період найбільшого застосування мов 1-го покоління – 60-70-ті роки.

До другої категорії відносять дані, які програміст має право сам створити, об'єднавши кілька базових типів даних. Головна відмінність від даних першої категорії полягає в тому, що в цьому об'єднанні можна використати різні базові типи даних. В результаті утворюються конструйовані типи даних. Приклади таких типів даних: структури (struct) і об'єднання (union) в мові С або записи (record) в мові Pascal. Ті мови, які містять,

окрім базових, ще й конструйовані типи даних, називаються мовами програмування 2-го покоління. Найбільш відомі мови 2-го покоління – C, Pascal, PL/1. Період найбільшого застосування мов 2-го покоління – 70-80-ті роки.

До третьої категорії відносять дані, які програміст має право створити, додавши в конструйовані типи також і операції над даними . В результаті утворюються абстрактні типи даних, в яких можуть бути об'єднані кілька базових типів даних і введені функції з цими базовими типами. Приклади таких типів даних: класи (class) в мові C++ або об'єкти (object) в мові Object Pascal. Ті мови, які містять, окрім базових і конструйованих, ще й абстрактні типи даних, називаються мовами програмування 3-го покоління. Найпершими представниками мов 3-го покоління стали об'єктні мови Simula-67 та SMALLTALK. Згодом абстрактні типи даних були введені в універсальні мови програмування, як у нові, так і у вже існуючі мови 2-го покоління. Тому мови 3-го покоління також називають об'єктно-орієнтованими. До таких мов відносять: ADA, C++, Object Pascal, Modula-2, CLU. Період найбільшого застосування мов 3-го покоління – 80-90-ті роки.

До четвертої категорії відносять дані, які програміст має право створити, додавши в абстрактні типи даних нові можливості роботи над даними. Якщо абстрактний тип даних характеризується двома параметрами (дані і функції над ними), то тип даних четвертої категорії характеризується вже трьома параметрами: властивостями, подіями та методами. Методи – це ті ж функції, властивості є подальшим розвитком параметра "дані", а події – це вже новий параметр. В результаті утворюються компонентні типи даних. Приклади таких типів даних: керуючі елементи в мові Visual Basic або компоненти в програмних пакетах Visual C++, C++ Builder та Delphi. Ті мови, які містяться в останніх трьох пакетах (C++ та Object Pascal) , можна назвати мовами програмування 4-го покоління. Ці мови програмування, незважаючи на однакові назви з мовами 3-го покоління, мають дуже багато нових характеристик, що дає право віднести їх вже до нового покоління. До 4-го покоління можна віднести також ще кілька зовсім нових мов програмування, орієнтованих на роботу в комп'ютерних мережах, наприклад мову Java. Варто також відмітити, що обов'язковим атрибутом мов 4-го покоління часто вважають також можливість роботи зі стандартними базами даних. Період початку широкого застосування мов 4-го покоління – починаючи з 90-тих років і до нашого часу.

Цікаво провести спільний аналіз мов різних поколінь із стилями програмування. Зовсім неважко помітити майже повний збіг одного стилю програмування – традиційного, структурного, об'єктно-орієнтованого та компонентного – відповідному поколінню мов програмування.

1 Коротке знайомство з програмним пакетом Visual C++ 6.0

1.1 Створення проектів в Visual C++ 6.0

Стиль програмування, прийнятий в програмному пакеті Visual C++ 6.0, є подальшим розвитком модульного програмування, коли спочатку розробляються незалежні один від одного модулі, окремо компілюються, а потім об'єднуються разом в один виконуваний файл. Сучасне модульне програмування базується на концепції проекту, який включає в себе велику кількість різних файлів: вихідних файлів програмного коду на мові C++, ресурсних файлів, довідкових файлів і т.д. В процесі компіляції автоматично утворюється ще цілий ряд допоміжних файлів. І лише на етапі редагування ми отримуємо єдиний файл або типу *.exe (завантажувальний файл програмного коду) або типу *.dll (завантажувальний файл динамічної бібліотеки). Особливістю пакету Visual C++ являється можливість зв'язування декількох проектів в межах єдиної робочої області. Для комфорту роботи робоча область утворюється завжди, навіть якщо буде використовуватися лише один проект. В табл.1.1 представлені основні типи файлів, які використовуються в проектах і робочих областях.

Таблиця 1.1 - Основні типи файлів пакету Visual C++

Типи файлів	Призначення файлів
*.cpp	Основний файл для програмного коду
*.h	Заголовний файл для програмного коду або ресурсів
*.rc	Основний ресурсний файл
*.rc2	Файл ресурсів, які не редагуються у Visual C++
*.ico	Файл для збереження піктограм ресурсів
*.bmp	Файл для збереження зображень ресурсів
*.def	Файл визначень модуля
*.obj	Об'єктний файл програмного коду
*.res	Об'єктний файл ресурсів
*.exe	Завантажувальний файл програмного коду
*.dll	Завантажувальний файл динамічної бібліотеки
*.clw	Інформаційний файл для роботи з класами
*.dsp	Файл проекту
*.dsw	Файл робочої області

При першому знайомстві з середовищем Visual C++ нелегко розібратися з такою великою кількістю файлів. Тут на допомогу може прийти один із Майстрів прикладних програм. Викликавши одного із таких Майстрів, і послідовно даючи відповіді на його запитання, можна швидко створити нескладну програму, наприклад, знамениту *Hellow World.*

І все ж вивчення програмування в середовищі Visual C++ краще розпочинати не з Майстрів. Написавши самостійно свою власну програму ви, по-перше, зможете розібратися у всіх тонкощах програмування, по-друге, зможете легко вносити зміни у таку програму і, нарешті, отримаєте меншу кількість вихідних файлів, ніж при використанні Майстра.

Розпочнемо підготовку до створення найпростішої програми в пакеті Visual C++. Перед тим як запустити на виконання цей пакет, необхідно підготувати папку де будуть зберігатися ваші файли. Нехай, наприклад, це буде папка *Visual++* на диску D.

Після першого запуску Visual C++ з'являється головне вікно інтегрованого середовища розробки (IDE) приблизно такого вигляду, як показано на рис.1.1

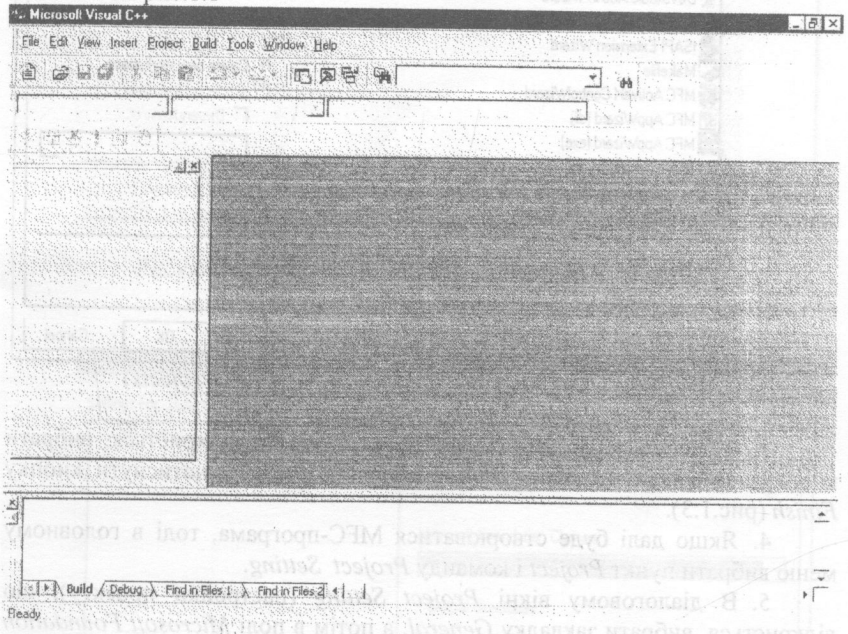


Рисунок 1.1 - Головне вікно Visual C++ 6.0

Для того, щоб створити новий проект, необхідно виконати наступні кроки:

1. В головному меню вибрати пункт *File*, задати команду *New...*, а в діалоговому вікні *New*, яке відкриється, вибрати вкладку *Projects* (як правило, ця вкладка спочатку завжди відкрита) (рис.1.2).
2. В цьому діалоговому вікні необхідно задати такі параметри:

- в полі *Location* вказати адресу робочої папки проекту, наприклад "*D:\VisualC++\Project*
- в полі *Project name* задати ім'я проекту, наприклад *MyProject*;

- в списку типів проекту вибрати тип *Win32 Application*;
- натиснути кнопку *Create new workspace*;
- вибрати платформу, для якої створюється проект: *Win 32*.
- закрити діалогове вікно *New*.

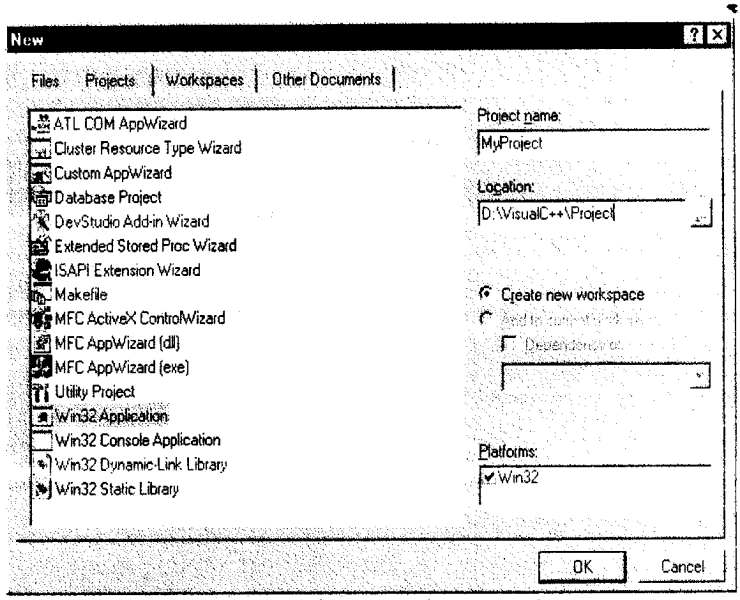


Рисунок 1.2 - Діалогове вікно *New* (вкладка *Projects*)

3. В діалоговому вікні *Win32 Applicition*, що відкриється, вибрати кнопку-перемикач *An empty project* (пустий проект) і натиснути кнопку *Finish* (рис.1.3).

4. Якщо далі буде створюватися MFC-програма, тоді в головному меню вибрати пункт *Project* і команду *Project Setting*.

5. В діалоговому вікні *Project Setting* (настройки проекту), що відкриється, вибрати закладку *General*, а потім в полі *Microsoft Foundation Classes* вибрати рядок *Use MFC in Static Library* (рис.1.4). Закрити діалогове вікно.

6. В пункті *Project* вибрати команду *Add to Project* (Додати до проекту).

Якщо програма буде набиратися з клавіатури, тоді в підпункті меню, що з'явився, вибрати команду *New...* Після цього з'явиться вікно текстового редактора, де і можна набирати потрібну програму.

Якщо файли програми попередньо підготовлені, тоді в підпункті меню, що з'явився, вибрати команду *Files...* В діалоговому вікні, яке з'яви-

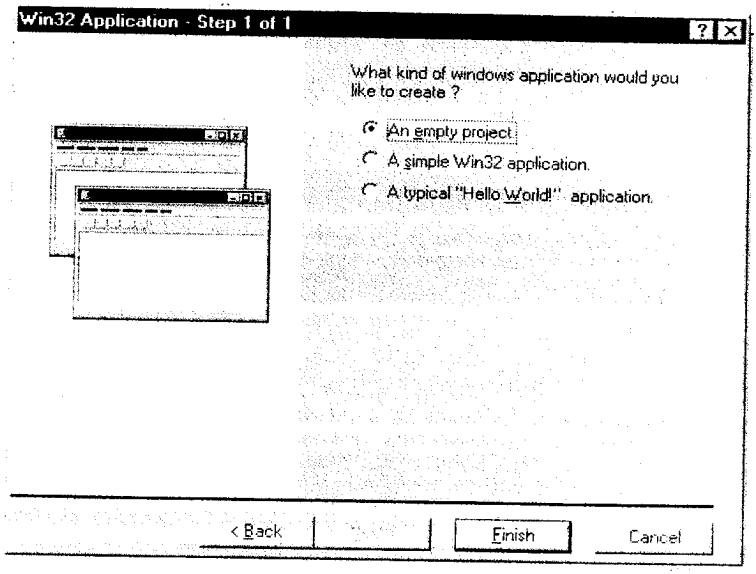


Рисунок 1.3 - Діалогове вікно *Win32 Applicition*

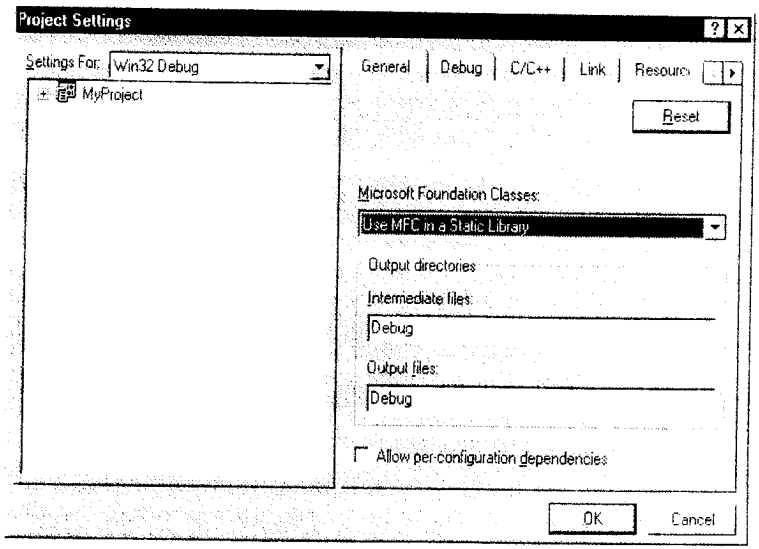


Рисунок 1.4 - Діалогове вікно *Project Setting*

лося, вибрати потрібні файли для включення в створюваний проект, і закрити діалогове вікно (якщо файли програми були записані в папку *D:\VisualC++\Project*, тоді для знаходження цих файлів достатньо лише піднятися на один рівень вверх по ієрархії файлів проекту).

Якщо проект складається лише з одного файлу початкової програми на C++ (типу *.cpp) і необхідно створювати якийсь ресурс (наприклад, меню), тоді необхідно виконати такі дії.

Нехай проект вже створений і відкритий.

1. В головному меню вибрати пункт *File*, задати команду *New...*, а в діалоговому вікні *New*, яке відкриється (рис.1.5), вибрати вкладку *Files*.
2. В цьому діалоговому вікні необхідно вказати такі параметри:

- в полі *Location* вказати адресу проекту (як правило, достатньо лише піднятися на один рівень вверх по ієрархії файлів проекту);
- в полі *Name* вказати ім'я ресурсного файлу;
- в списку типів файлів вибрати тип *Resours Script*;
- встановити перемикач в положення *Add to Project*.

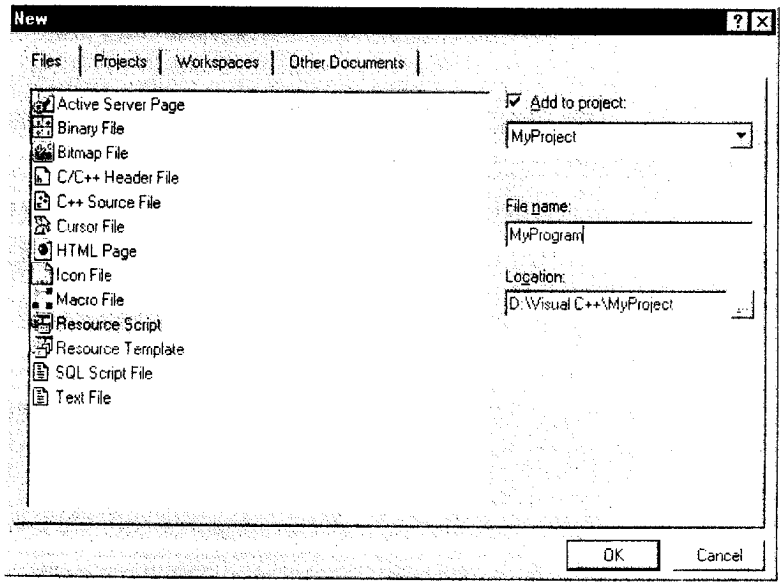


Рисунок 1.5 - Діалогове вікно *New* (вкладка *Files*)

В результаті до проекту додаються, щонайменше два нових файли: ресурсний файл (файл типу *.rc) і файл ідентифікаторів ресурсів Resource.h. Останній файл необхідно ще вручну підключити до проекту.

Далі необхідно викликати редактор конкретного ресурсу для створення цього ресурсу таким чином.

1. В головному меню вибрати пункт *Insert* і команду *Resource*.
2. В діалоговому вікні *Insert Resource*, що з'являється (рис.1.6), вибрати тип створюваного ресурсу, наприклад *Menu*.
3. Натиснути кнопку *New* для виклику редактора даного ресурсу.

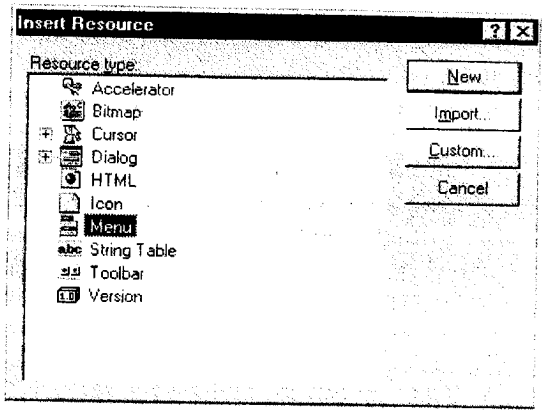


Рисунок 1.6 - Діалогове вікно *Insert Resource*

1.2 Компіляція, компонування, виконання і налагодження програми

Найшвидшим способом запуску проекту на виконання є натиснення на клавіатурі функціональної клавіші F5. Після підтвердження запиту системи, починаються процеси компіляції, компонування і виконання файлів проекту. При відсутності помилок в програмі ми отримаємо очікуваний результат на екрані дисплея.

Однак, на практиці рідко вдається відразу отримати робочу програму, особливо у випадку великих проектів. Тому корисно розібратися з процесами налагодження і пошуку помилок в програмі.

Згідно з термінологією розробників фірми Microsoft, процес створення виконувального файлу програми називається побудовою (build) програми. Тому в пункті *Build* головного меню є команди:

Build/Compile – компіляція програми;

Build/Build – створення завантажувального модуля;

Build/Rebuild All – повторне компонування завантажувального модуля без перевірки дати створення файла проекту;

Build/Execute – виконання завантажувального модуля (якщо був змінений початковий текст програми, тоді перед виконанням вона буде заново відкомпільована і скомпонована).

Розрізняють три типи помилок:

- синтаксичні помилки;
- помилки виконання програми;
- логічні помилки.

Синтаксичні помилки виявляються на етапі компіляції програми і відображаються у вікні повідомлень *Output* в нижній частині екрана дисплея. Кожне повідомлення про помилку включає ім'я файлу, де вона була виявлена; номер рядка у файлі і тип помилки (*warning* – попередження, *error* – помилка, котра потребує виправлення). Для швидкого переходу від вікна повідомлень до вікна редагування для виправлення помилки, достатньо лише помістити курсор на рядку повідомлення, яке вас зацікавило, і натиснути клавішу [*Enter*].

Більш неприємними є логічні помилки і помилки виконання програми. Вони часто призводять до "зависання" або до аварійного завершення програми. Саме для виявлення таких помилок існують у Visual C++ потужні засоби налагодження.

Найбільш поширеним методом налагодження програми є її покрокове виконання. Для цього передбачені такі основні команди в пункті *Debug* головного меню:

1) *Go ([F5])* – команда виконує програму до першої точки зупинки або повністю при відсутності точки зупинки.

2) *Restart ([Ctrl]+[Shift]+[F5])* – перезапуск програми на виконання спочатку.

3) *Break* – зупинка виконання програми. В подальшому виконані програми може бути продовжено, починаючи з даної позиції.

4) *Step Into ([F11])* - виконання програми по рядках із входженням в функції, що викликаються.

5) *Step Over ([F11])* - виконання програми по рядках без входження в функції, що викликаються.

6) *Run to Cursor ([Ctrl]+[F10])* – виконання програми до поточної позиції курсора.

7) *Step into Specific Function* – покрокове виконання вибраної функції.

Точки зупинки можуть бути введені в програму різними способами. Найпростіший полягає в тому, що курсор встановлюється на рядок, на якому потрібно зупинити виконання програми, і натискається функціональна клавіша *[F9]*. Інший спосіб полягає у виклику діалогового вікна *Breakpoints* командою *Breakpoints* із пункту *Edit* головного меню. У вказаному вікні необхідно або набрати номер рядка в

полі *Break at*. Можна також використати контекстне меню, в якому необхідно вибрати команду створення точки зупинки в тому рядку, де в даний момент у вікні редагування знаходиться курсор.

При будь-якому способі рядок з точкою зупинки буде помічений у вікні редагування червоним кружечком в крайній лівій позиції.

Вилучити точку зупинки також можна декількома способами. Наприклад, за допомогою діалогового вікна *Breakpoints*, або командою *Remove Breakpoints* з контекстного меню (курсор заздалегідь повинен знаходитися на відповідному рядку тексту програми).

У Visual C++ передбачені також спеціальні заходи налагодження *Windows*-програм. Налагоджувальну інформацію про вікна, процеси, точки і повідомлення можна отримати від спеціального інструмента *Spy++*, який можна викликати з пункту *Tools* головного меню. У відкритому вікні *Spy++* необхідно вибрати відповідну команду (*Windows*, *Processes*, *Threads*). Потім в новому вікні перегляду вибрати конкретне вікно, процес або потік, і через контекстне меню (клацнувши правою кнопкою миші) вказати характеристики, що Вас зацікавили (наприклад, переглянути повідомлення, які направляються у вікно).

1.3 Особливості стилю програм в середовищі Windows

Можливості будь-якої мови програмування в першу чергу визначаються використовуваними типами даних. В програмах мовами C та C++ в середовищі Windows підтримуються стандартні типи даних C та C++, а також з'явилось багато нових типів даних. В табл. 1.2 наведені найбільш популярні нові типи даних.

Ясність та доступність програм для аналізу в значній мірі залежить від способу позначення змінних. Ознакою професіоналізму є використання мнемоніки з кількох слів і виділенням окремих слів великими буквами. Тоді без коментарів буде зрозуміло, що, наприклад, змінна ArraySize позначає розмір масиву.

Така ясність опису типів та імен змінних веде свій початок від Паскалю. Перша версія Windows, тоді ще оболонки, була реалізована саме на цій мові програмування. При переході Windows із Паскалю на Сі залишилося багато позитивних сторін Паскалю.

Потім з'явились нові цікаві ідеї. Дуже цінною ідеєю, що швидко отримала визнання, стало включення в ім'я змінної кількох початкових букв (префікса), які позначають її тип. Оскільки це вперше було запропоновано програмістом фірми Microsoft Чарльзом Сімонаї, угорцем за національністю, тому такий спосіб нотації отримав назву угорської нотації. Наприклад, відразу ясно, що змінна lpszClassName є покажчиком далекого типу на рядок, який закінчується нуль-символом. В табл. 1.3 наведені префікси для найбільш поширених типів даних.

Таблиця 1.2 - Нові типи даних в мовах С та С++

Тип даних	Опис
BYTE	Однобайтове беззнакове ціле число
BOOL	Логічна змінна, яка приймає значення TRUE (вірно) і FALSE (невірно)
WORD	32-розрядне беззнакове ціле
DWORD	Беззнакове ціле подвійної довжини
LONG	32-розрядне ціле зі знаком
LPSTR	32-розрядний покажчик на рядок
LPCSTR	Те ж саме, що й *LPSTR*, але покажчик тільки на константні рядки
UINT	32-розрядне беззнакове ціле число
WCHAR	16-розрядний символ UNICODE
LPVOID	Узагальнений тип покажчика, еквівалентний VOID
HDC	Дескриптор контексту пристрою
HWND	Дескриптор вікна
HANDLE	32-розрядне беззнакове ціле, що використовується як дескриптор
LPARAM	Використовується для опису молодшого аргументу повідомлення
WPARAM	Використовується для опису старшого аргументу повідомлення
WINAPI	Заміняє специфікацію FAR PASCAL в описі API-функцій
CALLBACK	Заміняє специфікацію FAR PASCAL в функціях з поверненням
MSG	Структура, яка визначає параметри повідомлення
WNDCLASS	Структура, яка визначає клас вікна
PAINTSTRUCT	Структура, що визначає область вікна, яке потрібно перерисувати

Таблиця 1.3 - Префікси для WINDOWS

Префікс	Визначення на С	Пояснення
a		Масив
b	BOOL	int
by	BYTE	unsingned char
c	char	
n	short або int	
i	int	
w	WORD	unsingned int
h	HANDLE	unsingned int
h	HWND	unsingned int
l	LONG	long
dw	DWORD	unsingned long
s		Рядок
sz		Рядок, що закінчується нулем
fn	function	Функція
p	*	Покажчик
lp	far *	Дальній покажчик
np	near *	Ближній покажчик
x	short	При використанні як координати X або розміру
y	short	При використанні як координати Y або розміру

2 Створення прикладних API-програм в середовищі Windows

2.1 Структура мінімальної прикладної API-програми

Найпростіша прикладна програма, яка написана для виконання в середовищі Windows, завжди містить не менше двох функцій: головну функцію WinMain() і віконну функцію.

Далі будуть розглядатись також стандартні функції Windows, які об'єднуються у велику групу, під спільною назвою – API-функції. Прикладну програму в середовищі Windows, в якій використовуються API-функції, будемо в подальшому також коротко називати - API-програмою.

Функції WinMain() передається керування в початковий момент завантаження програми і нею ж і закінчується виконання програми. Ця функція по суті виконує ту ж роль, що і функція main() в звичайних програмах на C і C++.

Функція WinMain() виконує такі завдання:

1) визначення класу вікна;
2) реєстрація класу вікна;
3) створення вікна;
4) відображення і оновлення вікна;
5) створення циклу обробки черги повідомлень;
6) завершення роботи програми;

Таким чином, спочатку визначається клас вікна, тобто задаються найбільш загальні властивості вікон: стилі, заголовки, піктограми, курсори, меню тощо. Всі ці параметри відповідають полям структури *WNDCLASS*. Для найпростішої програми для Windows багато із цих полів мають нульове значення. З ускладненням програми (наприклад, включення меню), відповідні поля структури будуть конкретизуватись.

Після визначення всіх необхідних значень полів структури *WNDCLASS* необхідно зареєструвати клас вікна за допомогою API-функції *RegisterClass*(). Якщо реєстрація вікна пройшла успішно, то функція повертає значення *TRUE*, в протилежному випадку (наприклад, у випадку нестачі пам'яті), функція повертає *FALSE*, після чого виконання програми закінчується.

Реєстрація класу вікна ще не свідчить про створення самого вікна. Вікно утворюється в результаті виклику API-функції *CreateWindow*(). Ця функція на основі раніш заданих загальних характеристик утворює конкретний об'єкт вікна, який має свій заголовок, розміри, координати початку та інші дані. Якщо деякі з параметрів вікна задати нульовими, тоді Windows буде надавати ці значення за своїм бажанням. В результаті, наприклад, при кожному запуску програми на виконання, її вікно буде розміщуватися кожного разу в новому місці екрану.

Створення конкретного об'єкту вікна ще не буде означати, що ми його відразу побачимо на екрані. Для відображення вікна на екрані

служить API-функція *ShowWindow*(), а для оновлення вікна – API-функція *UpdateWindow*().

Всі перераховані вище дії функція *WinMain*() виконує відразу після запуску програми на виконання. Потім, протягом всього часу, поки програма активна, функція *WinMain*() забезпечує зв'язок програми з "зовнішнім світом" шляхом обробки повідомлень, які надходять з черги програми. Для виконання цього важливого завдання існує стандартний цикл обробки повідомлень, який в найпростішому варіанті складається із таких функцій.

API-функція *GetMessage*() вибирає поточне повідомлення із черги і передає його API-функції *TranslateMessage*(), яка перекладає усі повідомлення від клавіатури, у відповідності із стандартом *ANSI*. Потім API-функція *DispatchMessage*() передає повідомлення для обробки у віконну функцію.

Другою обов'язковою функцією в будь-якій програмі в середовищі Windows є віконна функція. Вона може мати довільне ім'я, назвемо її, наприклад *WindowFunc*(). Віконна функція отримує повідомлення від головної функції *WinMain*(), і тому вона повинна проводити їх обробку.

В якості параметрів віконної функції передаються дескриптор вікна *hWnd*, з яким пов'язана *WindowFunc*(), саме повідомлення *iMessage* і два додаткових параметри *wParam* і *lParam*, які, як правило, служать для уточнення повідомлення, що надійшло. Ці чотири вхідні параметри, а також тип значення, що повертається, завжди тісно пов'язані і не можуть бути змінені.

Для обробки повідомлень у віконній функції є оператор *switch* з відповідними гілками *case* або набір макросів, які пов'язують повідомлення з необхідними функціями обробки.

Під час активного функціонування програми в віконну функцію може бути направлена велика кількість різних повідомлень, однак, зовсім необов'язково на всі повідомлення реагувати. Обробляються програмно за допомогою власних функцій лише ті повідомлення, які мають практичний сенс для даної програми. Всі інші повідомлення, які отримуються віконною функцією, викликом API-функції *DefWindowProc*() повертаються у *Windows* для стандартної обробки.

Однак, навіть в самому мінімальному варіанті віконна функція повинна самостійно оброблювати повідомлення WM_DESTROY, яке посилається при завершені програми. Оброблюючи його, віконна функція повинна викликати API-функцію *PostQuitMessage*(). В результаті прикладній програмі посилається повідомлення *WM_QUIT*, отримавши яке, API-функція *GetMessage*() і завершує виконання програми.

Відносно віконної функції необхідно пам'ятати також таке:

- віконна функція викликається безпосередньо операційною системою *Windows*, а функція *WinMain*() не викликає її, а лише посилає повідомлення;

- ім'я віконної функції повинно бути зареєстровано в *Windows* шляхом запису її імені при визначенні класу вікна в функції *WinMain*():

WndClass.lpfnWndProc=WindowFunc;
WndClass.cbClsExtra=NULL;

Таким чином, мінімальна API-програма буде мати такий вигляд.

```
// win_min.cpp
#include <windows.h>
#include <stdio.h>
#include <string.h>
/*   Прототип віконної функції     */
LRESULT CALLBACK WindowFunc(HWND,UINT,UINT,LPARAM);

/*   Головна функція в Windows 95/98/ME   */
int WINAPI WinMain(HINSTANCE hInstance,
HINSTANCE hPrevInstance, LPSTR lpszCmdLine,int nCmdShow)
 {
       WNDCLASS WndClass;
       HWND hWnd;
       MSG msg;
/*    Ім'я класу вікна    */
 WndClass.lpszClassName=(LPSTR)"MyClass";
/*    Адреса віконної функції    */
 WndClass.lpfnWndProc=WindowFunc;
/*    Дескриптор даного екземпляра програми    */
 WndClass.hInstance=hInstance;
/*    Додаткові дані для класу          */
 WndClass.cbClsExtra=NULL;
/*    Додаткові дані для вікна    */
 WndClass.cbWndExtra=NULL;
/*    Дискриптор піктограми    */
 WndClass.hIcon=LoadIcon(NULL,IDI_APPLICATION);
/*    Дискриптор курсора    */
 WndClass.hCursor=LoadCursor(NULL,IDC_ARROW);
/*    Колір фону вікна    */
 WndClass.hbrBackground=(HBRUSH)GetStockObject(WHITE_BRUSH);
/*    Ім'я (ідентифікатор) головного меню    */
 WndClass.lpszMenuName=(LPSTR)NULL;
/*    Тип вікна    */
 WndClass.style=NULL;

          if(!RegisterClass(&WndClass))
            return NULL;
/*    Створення вікна    */
 hWnd=CreateWindow("MyClass",      /*    Ім'я вікна                    */
```

```
    "MyProgram",                /*   Заголовок вікна                    */
    WS_OVERLAPPEDWINDOW,        /*   Тип вікна (стандартне)             */
    CW_USEDEFAULT,              /*   X-координата                       */
    CW_USEDEFAULT,              /*   Y-координата                       */
    CW_USEDEFAULT,              /*   Ширина вікна                       */
    CW_USEDEFAULT,              /*   Висота вікна                       */
    NULL,                       /* Дескриптор екземпляра-предка вікна*  /
    NULL,                       /*   Дескриптор головного меню          */
    hInstance,                  /*   Дескриптор екземпляра програми     */
    NULL);                      /*   Додаткова інформація               */
    if( ! hWnd)
        return NULL;

ShowWindow(hWnd,nCmdShow);      /*   Функція відображення вікна         */
UpdateWindow(hWnd);             /*   Функція оновлення вікна            */

/*    Цикл обробки повідомлень*/
 while(GetMessage(&msg, NULL, NULL, NULL))
 {
/*    Трансляція віртуальних кодів клавіш в клавіатурні повідомлення   *
      TranslateMessage(&msg);
/*    Повернення повідомлення назад в Windows   */
      DispatchMessage(&msg);
 }
      return msg.wParam;
}

/*   Віконна функція в Windows 95/98   */
LRESULT CALLBACK WindowFunc(HWND hWnd,UINT iMessage,
                            WPARAM wParam,LPARAM lParam)
{
/*   Цикл обробки черги повідомлень   */
     switch(iMessage)
     {
                  break;
     case WM_DESTROY:
      PostQuitMessage(0); break;
     default:
     return(DefWindowProc(hWnd,iMessage,wParam,lParam));
     }
          return 0L;
}
```

Результатом виконання наведеної вище API-програми буде поява на екрані дисплея вікна програми (рис. 2.1).

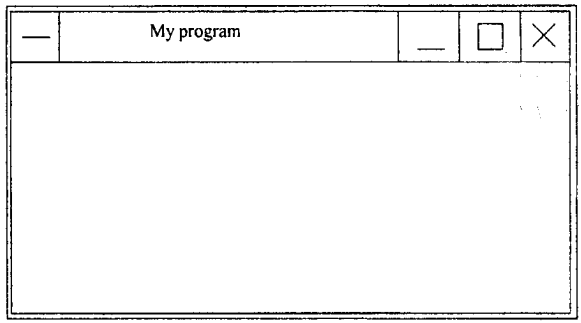


Рисунок 2.1 - Головне вікно мінімальної API-програми

2.2 Векторна графіка в API-програмах

Програмування графіки в Windows–програмах досить суттєво відрізняється від програмування в однозадачних операційних системах типу MS-DOS. Тому детально зупинимось на всіх вузлових питаннях, які визначають особливості програмування в API-програмах: контекст пристрою, формування кольорів, логічні інструменти графіки та використання найбільш поширених API-функцій векторної графіки.

2.2.1 Контекст пристрою

В програмах, які написані для виконання в операційній системі MS-DOS, для даних, що виводяться, необхідно вказати конкретний фізичний пристрій виведення (дисплей, принтер і т.д.). Зовсім інша ситуація при програмуванні в середовищі Windows. В Windows-програмах не вказується конкретний вихідний пристрій, а звертання йде до контексту пристрою.

Контекст пристрою (device context (dc)) – це структура, яка зв'язує програму з драйвером пристрою виведення. Тому перед початком виведення інформації необхідно за допомогою спеціальних функцій *API* одержати контекст пристрою, тобто встановити зв'язок з конкретним пристроєм виведення. Якщо мова йде про дисплей, тоді часто потрібні також додаткові відомості про область екрану (робоча область вікна, весь екран і т.д.), куди будуть виводитися дані.

Після закінчення процесу виведення програма повинна звільнити контекст пристрою за допомогою відповідних функцій *API*. Цю операцію необхідно виконувати, оскільки в Windows одночасно може існувати

обмежена кількість контекстів пристроїв, і через деякий час в системі може не знайтися вільних ресурсів для представлення поточного контексту пристрою.

Контекст пристрою має також і ряд інших корисних властивостей.

Розглянемо детальніше функції API для роботи з контекстами пристроїв.

Найбільш універсальною парою функцій API для одержання і звільнення контексту пристрою служать відповідно API-функції *GetDC*() і *ReleaseDC*(), які мають такий формат:

HDC GetDC (HWND hWnd);
int Release(HWND hWnd, HDC hdc);

GetDC() повертає контекст пристрою, асоційованого з вікном, дескриптор якого задається параметром *hWnd*. API-функція *ReleaseDC*() повертає ненульове значення, якщо контекст пристрою дійсно був звільнений. Параметр *hWnd* задає дескриптор вікна, який асоційований з контексту пристрою, що звільняється, а *hdc* – дескриптор звільненого контексту пристрою.

Ще однією широко використовуваною парою API-функцій для одержання і звільнення контексту пристрою служать відповідно функції *BeginPaint*() і *EndPaint*(), які мають такий формат:

HDC BeginPaint(HWND hWnd, PAINTSTRUCT &PaintStruct);
EndPaint(HWND hWnd, PAINTSTRUCT &PaintStruct); .

Важливою відмінністю цієї пари функцій від попередньої є її зв'язок з механізмом повідомлень *WM_PAINT*.

Тут необхідно зупинитися ще на одній характерній особливості програмування в *Windows*. Система не займається збереженням вікна прикладної програми. Якщо вікно програми буде звернуто в піктограму або перекрито іншим вікном, тоді при відновлені початкового вікна його вміст буде загублено. *Windows* не перерисовує вікно прикладної програми автоматично, а завдання перерисовування покладається на саму програму. Кожен раз, коли виникає задача перерисовування вікна, у віконну функцію програми надходить повідомлення *WM_PAINT* (це повідомлення посилається програмі також при створенні і першому відображенні вікна). Для обробки цього повідомлення необхідно одержати контекст пристрою з допомогою функції API *BeginPaint*(). Другим параметром цієї функції є покажчик на структуру *PAINTSTRUCT*, яка і задає координати прямокутної області в вікні, що підлягає перерисовуванню.

Ця структура визначається таким чином.

```
typedef struct tagPAINTSTRUCT
{
HDC hdc;            /* Дескриптор контексту пристрою          */
BOOL fErase;        /* Вірно, якщо перерисовується вміст вікна */
RECT rcPaint;       /* Координати області перерисовування     */
```

BOOL fRestore;
BOOL fIncUpdate;
BYTE rgbReserved[32];
} PAINTSTRUCT; .

В деяких випадках виникає задача зміни деякої області вікна. В цьому випадку корисною може стати API-функція *InvalidateRect*(), яка має такий формат:

BOOL *InvalidateRect(HWND hWnd, CONST RECT *lprc,*
BOOL fErase); .

Параметр *lprc* містить координати частини клієнтської області, яка повинна бути анульована (якщо він рівний NULL, тоді буде анульована вся клієнтська область). Якщо параметр *fErase* дорівнює TRUE, тоді анульована область вікна буде стерта.

Якщо необхідно анулювати непрямокутну область, тоді можна використати API-функцію *InvalidateRgn*().

2.2.2 Логічна система координат

Для виведення графіки використовується логічна система координат. За замовчуванням область виведення збігається з робочою областю вікна, верхній лівий кут робочої області вікна має нульові координати (*x=0, y=0*) і в якості логічних одиниць вимірювання координат використовуються пікселі (одна логічна одиниця дорівнює одному пікселю). Однак програмно можна змінювати співвідношення логічних і фізичних одиниць, а також змінювати розміри області виведення.

Для встановлення поточного режиму відображення використовується API-функція *SetMapMode*():

int SetMapMode(HDC hdc, int mode); .

Серед значень, які може приймати *mode* варто відмітити такі:

MM_ANISOTROPIC – програмно задається різноманітне масштабування по осях координат;

MM_LOMETRIC – одна логічна одиниця дорівнює 0,1 міліметра;

MM_LOENGLISH – одна логічна одиниця дорівнює 0,01 дюйма;

MM_TEXT – одна логічна одиниця дорівнює 1 пікселю;

MM_ISOTROPIC – логічні одиниці по осях X і Y мають однаковий фізичний розмір.

Розміри вікна в логічних одиницях можна задати з допомогою API-функції *SetWindowExtEx*():

BOOL SetWindowExtEx(HDC hdc, int x, int y, LPSIZE size);

Тут *x* і *y* – розміри вікна по горизонталі і вертикалі. Попередні розміри вікна копіюються в структуру *SIZE*, а якщо в цьому немає потреби, тоді останній параметр дорівнює *NULL*.

Початок області виведення можна змінити за допомогою API-функції *SetViewportOrgEx*():

BOOL SetViewportOrgEx(HDC hdc, int X, int Y, LPPOINT old Org); .

Тут *X* і *Y* – координати нової точки початку області виведення. Координати попереднього початку області виведення записуються в структуру *POINT*. Якщо попередні координати не потрібно зберігати, тоді останній параметр рівний *NULL*.

2.2.3 Формування кольорів

Існує три способи задання кольорів в API-програмах:

- використання *RGB* - значень;
- задання індексу кольору в логічній палітрі;
- знаходження *RGB* - значень відносно поточної палітри.

Найчастіше використовується перший спосіб, тому розглянемо його більш детально. В цьому випадку колір визначається 32–бітовим цілим типу *COLORREF*, яке зберігає макрос *RGB*:

COLORREF RGB(BYTE Red, BYTE Creen, BYTE Blue); ,

де *Red, Creen* і *Blue* – числа від 0 до 255, які визначають інтенсивність відповідно червоного, зеленого і синього кольору.

RGB–значення дозволяє задавати як окремі базові кольори максимальної інтенсивності, так і довільні комбінації інтенсивності базових кольорів. Наприклад:

RGB(255, 0, 0) – яскраво–червоний колір;

RGB(128, 128, 0) – жовтий колір;

RGB(255, 255, 255) – білий колір;

RGB(0, 0, 0) – чорний колір.

Задання кольору з допомогою *RGB*–значень використовується в якості параметрів багатьох API-функцій . Наприклад колір фону може бути встановлений з допомогою API-функції *SetBkColor*():

COLORREF SetBkColor(HDC hdc, COLORREF color); .

2.2.4 Перо та пензель

Всі графічні об'єкти в *Windows* рисуються за допомогою пера (*Pen*) і пензля (*Brush*). Розглянемо спочатку роботу з перами.

Наперед передбачені три вбудованих системних пера: чорне (*BLACK_PEN*), біле (*WHITE_PEN*) і невидиме (*NULL_PEN*). Ці пера можна одержати, викликавши API-функцію *GetStockObject()*:

HGDIOBJ GetStockObject(int object); .

Ця функція повертає дескриптор об'єкту *object*, якому присвоюється значення із набору системних пер, наприклад:

HPEN Pen=(HPEN) GetStockObject(WHITE_PEN); .

За замовчуванням використовується перо *BLACK_PEN*.

Всі стандартні пера рисують суцільні лінії товщиною рівно в один піксель. Для збільшення кількості пер і розширення їх функціональних можливостей служить API-функція *CreatePen()*:

HPEN CreatePen(int style, int width, COLORREF color); .

Ця функція дозволяє створити власне перо, яке дозволяє рисувати лінії заданого стилю (*style*), заданої товщини (*width*) і заданого кольору (*color*). Можливі стилі власного пера наведені в таблиці 2.1. Варто пам'ятати, що всі вказані стилі, окрім PS_SOLID, можливі лише для товщини лінії в один піксель (тобто для *width*=1).

Таблиця 2.1 - Стилі власного пера

Макрос	Тип лінії пера
PS_SOLID	Суцільна лінія
PS_DASH	Пунктирна лінія (рівномірні відрізки)
PS_DASHDOT	Штрих-пунктирна лінія
PS_DASHDOTDOT	Штрих-пунктирна лінія
PS_DOT	Точкова лінія
PS_INSIDEFRAME	Суцільна лінія всередині області, що обмежує
PS_NULL	Прозоре перо

Після створення перо необхідно пов'язати з контекстом пристрою за допомогою API-функції *SelectObject*():

HGDIOBJ SelectObject(HDC hdc, int object); .

Для зафарбовування замкнутих фігур система *Windows* використовує пензель, тобто бітовий зріз розміром 8x8 пікселів.

Як і у випадку з перами, існують вбудовані системні пензлі: суцільний білий (*WHITE_BRUSH*), суцільний чорний (*BLACK_BRUSH*), суцільний темно-сірий (*DKGRAY_BRUSH*), суцільний сірий (*GRAY_BRUSH*), суцільний світло-сірий (*LTGRAY_BRUSH*) і нульовий (*NULL_BRUSH*). Будь-який із них можна одержати з допомогою API-функції *GetStockObject()*, присвоївши параметру *object* одне із перерахованих вище значень. За замовчуванням використовується білий пензель.

При необхідності можна створити власний пензель. Найбільш часто потрібний суцільний пензель, який можна створити за допомогою API-функції *CreateSolidBrush*():

HPEN CreateSolidBrush(COLORREF color); .

Колір пензля, що утворюється, задається параметром *color*, а сама функція повертає дескриптор пензля.

Як і перо, пензель після створення необхідно пов'язати з контекстом пристрою за допомогою API-функції *SelectObject*().

Крім суцільного, можна також самостійно створити і інші типи пензлів:

- заштрихований (за допомогою API-функції *CreateHatchBrush*()): *HPEN CreateHatchBrush(int HatchStylle, COLORREF color)*; ,
- узорчатий (за допомогою API-функції *CreatePatternBrush*()).

Стиль штрихування задається параметром *HatchStylle* і може приймати одне із значень, вказаних в табл. 2.2.

Таблиця 2.2 - Стилі штрихування власного пензля

Макрос	Стиль штрихування
HS_BDIAGONAL	45-градусна штриховка зліва направо і зверху вниз
HS_CROSS	Горизонтальна і вертикальна штриховка хрест-навхрест
HS_DIAGCROSS	45-градусна штриховка хрест-навхрест
HS_FDIAGONAL	45-градусна штриховка зліва-направо і зверху вниз
HS_GORISONTAL	Горизонтальна штриховка
HS_VERTICAL	Вертикальна штриховка

При використанні узорчатого пензля спосіб заповнення замкнутої фігури визначається бітовою матрицею, яку можна створити в графічному редакторі.

Після закінчення роботи з власними перами і пензлями їх необхідно знищити за допомогою API-функції *DeleteObject*():

BOOL DeleteObject (HGDIOBY object); .

2.2.5 Рисування ліній

Щоб нарисувати одиночну пряму лінію, спочатку необхідно вказати її початкову точку, викликавши API-функцію *MoveToEx*():

BOOL MoveToEx(HDC hdc, int x, int y, LPPOINT lpPoint); .

Тут координати нової поточної позиції задаються параметрами x і y, а координати старої поточної позиції повертаються в структуру *POINT*. Якщо попередню позицію не потрібно зберігати, тоді цьому параметру присвоюється значення *NULL*.

Потім необхідно викликати API-функцію *LineTo*():

BOOL LineTo(HDC hdc, int x_1, int y_1); .

Ця функція рисує пряму від поточної позиції до точки з координатами x_1 і y_1.

Послідовність ліній, які з'єднуються між собою, можна також нарисувати з допомогою виклику API-функції *PolyLine*():

*BOOL PolyLine(HDC hdc, CONST POINT *lppt, int cPoints)*; ,

де *lppt* представляють собою масив структур типу *POINT*, який містить координати набору точок, а *cPoints* – кількість самих точок (яких повинно бути не менше двох). Наприклад, за допомогою наступного фрагмента програми можна нарисувати букву М:

```
POINT Points[5];
Points[0].x=50: Points[0].y=50;
Points[1].x=50; Points[1].y=150;
Points[2].x=100; Points[2].y=100;
Points[3].x=150; Points[3].y=50;
Points[4].x=150; Points[4].y=150;
PolyLine(HDC hdc, Points, 5);
```

Нарисувати дугу можна за допомогою API-функції *Arc*():

BOOL RoundRect(HDC hdc, int x_1, int y_1, int x_2, int y_2, int x_3, int y_3, int x_4, int y_4); .

Тут перші чотири параметри задають координати еліпса, параметри x_3 і y_3 – координати точки на початковому радіусі дуги, а параметри x_4 і y_4 – координати точки на кінцевому радіусі дуги.

2.2.6 Рисування стандартних геометричних фігур

Нарисувати прямокутник можна за допомогою API-функції *Rectangle*():

BOOL Rectangle(HDC hdc, int x_1, int y_1, int x_2, int y_2); .

Тут параметри x_1 і y_1 визначають координати лівого верхнього кута прямокутника, а параметри x_2 і y_2 – координати нижнього правого кута прямокутника (рис.2.2).

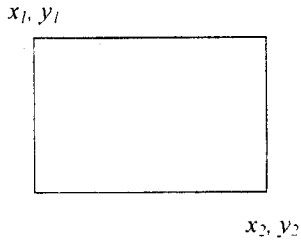


Рисунок 2.2 – Побудова прямокутника

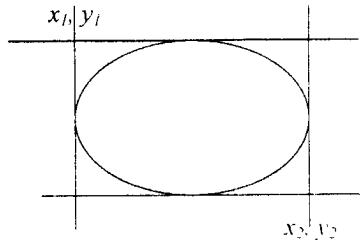


Рисунок 2.3 - Побудова еліпса

За допомогою API-функції *RoundRect*() можна нарисувати прямокутник із заокругленими кутами:

BOOL RoundRect(HDC hdc, int x_1, int y_1, int x_2, int y_2, int x_3, int y_3); .

Заокруглення кутів прямокутника визначається параметрами x_3 і y_3, які задають відповідно ширину і висоту еліпса, що визначає дуги.

Для побудови еліпса або круга використовується API-функція *Ellipse*():

BOOL Ellipse(HDC hdc, int x_1, int y_1, int x_2, int y_2); .

Еліпс визначається через прямокутник з координатами x_1, y_1, x_2, y_2, в який вписується еліпс (рис.2.3). Щоб отримати круг необхідно задати квадрат.

Для того, щоб нарисувати багатокутник з двома і більше вершинами, використовується API-функція *Polygon*():

*BOOL Polygon(HDC hdc, CONST POINT *lpPoints, int nCount);* .

Параметр *lpPoints* представляє собою масив структур *POINT*, які містять координати вершин, а *nCount* є числом точок в масиві (їх повинно бути не менше двох). Наприклад, для трикутника необхідно записати :

```
POINTS Points[3]={{25, 50},
                  {50, 25},
                  {75, 50}};
Polygon(hdc, Points, 3);
```

.

На відміну від подібної API-функції *PolyLine*(), API-функція *Polygon*(), автоматично з'єднує останню вершину в масиві з першою, і, тому, завжди рисує замкнуті фігури.

Для того, щоб нарисувати сегмент (фігуру, яка утворилася при перетині еліпса з прямою), існує спеціальна API-функція *Chord*():

BOOL Chord(HDC hdc, int x_1, int y_1, int x_2, int y_2, int x_3, int y_3,
int x_4, int y_4); .

Тут дві перші пари координат задають обмежувальний прямокутник еліпсу, третя пара координат є координатами точки на початковому радіусі сегмента, а четверта пара координат – точки на його кінцевому радіусі (рис. 2.4).

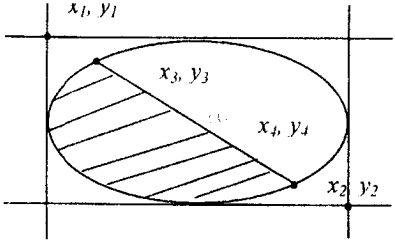


Рисунок 2.4 - Побудова сегмента

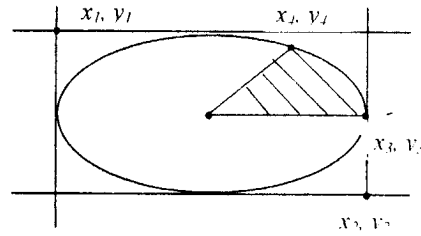


Рисунок 2.5 - Побудова сектора

Для рисування сектора призначена API-функція *Pie*():

BOOL Pie(HDC hdc, int x_1, int y_1, int x_2, int y_2, int x_3, int y_3, int x_4, int y_4); .

Тут також дві перші пари координат задають обмежувальний прямокутник еліпсу. Кінці сектора позначаються точками з координатами x_3, y_3 і x_4, y_4. Побудова фігури здійснюється проти годинникової стрілки (від точки x_3, y_3 до точки x_4, y_4) (рис.2.5).

Існує декілька спеціальних API-функцій для задання кольору при рисуванні геометричних фігур.

Колір окремого пікселя можна встановити за допомогою API-функції *SetPixel*():

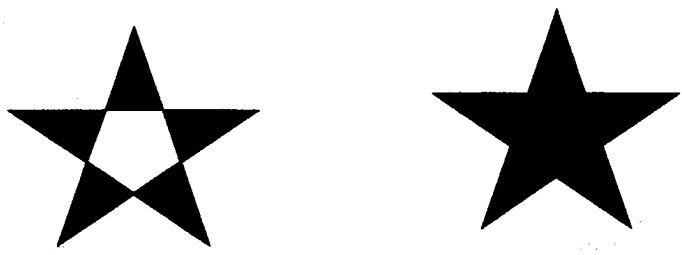
COLORREF SetPixel(HDC hdc, int X, int Y, COLORREF color); .

тут *X* і *Y* – координати точки, в якій необхідно встановити потрібний колір *color*.

Для складного багатокутника із самоперетинами, можна задати зафарбовувані області всередині нього за допомогою API-функції *SetPolyFillMode*():

int SetPolyFillMode(HDC hdc, int PolyFillMode);

Якщо параметр *PolyFillMode* має значення *ALTERNATE*, тоді заповнюються тільки ті внутрішні області, в які можна попасти зовні, перетинаючи межу непарне число раз. Якщо ж вказаний параметр має значення *WINDING*, тоді зафарбовуються всі внутрішні області. Класичним прикладом для ілюстрації роботи цієї функції є п'ятикутна зірка (рис.2.6).



а) б)

Рисунок 2.6 - Зафарбування замкнутої фігури за допомогою API-функції *SetPolyFillMode*():

а) якщо *PolyFillMode* має значення *ALTERNATE*.

б) якщо *PolyFillMode* має значення *WINDING*.

Як приклад, розглянемо віконну функцію, в якій програмується виведення кількох геометричних фігур. При натисненні лівої клавіші миші в робочій області вікна програми з'являться червоний прямокутник і жовтий трикутник із зеленою рамкою, а також синій еліпс з червоною рамкою, причому для зафарбування прямокутника використана вертикальна штриховка. При натисненні правої клавіші миші з'явиться слово *"TEST"*. Перед кожним виведенням вказаних даних робоча область вікна очищається білим кольором.

```
LRESULT CALLBACK WindowFunc(HWND hWnd,UINT iMessage,
                            WPARAM wParam,LPARAM lParam)
{
 HDC hdc;
HPEN hRedPen,hGreenPen;
HBRUSH hbrushRed,hbrushYellow,hbrushBlue,hbrushWhite;
POINT Points[3]={{ 430,350},
                 { 160,350},
                 { 430,190}};
char str[80]="TEST";
switch(iMessage)
{
 case WM_RBUTTONDOWN:
        hdc=GetDC(hWnd);
        hbrushWhite=CreateSolidBrush(RGB(255,255,255));
        SelectObject(hdc,hbrushWhite);
        Rectangle(hdc,0,0,600,400);

        SetTextColor(hdc,RGB(255,0,0));
        SetBkColor(hdc,RGB(0,255,255));
        TextOut(hdc,200,250,str,strlen(str));
        ReleaseDC(hWnd,hdc);
        break;

case WM_LBUTTONDOWN:
        hdc=GetDC(hWnd);
        hbrushWhite=CreateSolidBrush(RGB(255,255,255));
        SelectObject(hdc,hbrushWhite);
        Rectangle(hdc,0,0,600,400);

        hRedPen=CreatePen(PS_SOLID,10,RGB(255,0,0));
        SelectObject(hdc,hRedPen);
        hbrushBlue=CreateSolidBrush(RGB(0,0,255));
        SelectObject(hdc,hbrushBlue);
        Ellipse(hdc,50,50,350,200);
```

```
            hGreenPen=CreatePen(PS_SOLID,10,RGB(0,255,0));
            SelectObject(hdc,hGreenPen);
            hbrushYellow=CreateSolidBrush(RGB(255,255,0));
            SelectObject(hdc,hbrushYellow);
            Polygon(hdc,Points,3);

            hbrushRed=CreateHatchBrush(HS_VERTICAL,RGB(255,0,0));
            SelectObject(hdc,hbrushRed);
            Rectangle(hdc,50,250,150,350);

            DeleteObject(hRedPen);
            DeleteObject(hGreenPen);
            DeleteObject(hbrushRed);
            DeleteObject(hbrushBlue);
            DeleteObject(hbrushYellow);
            DeleteObject(hbrushWhite);
            ReleaseDC(hWnd,hdc);
             break;
        case WM_DESTROY:
         PostQuitMessage(0); break;
        default:
        return(DefWindowProc(hWnd,iMessage,wParam,lParam));
        }
            return 0L;
}
```

2.2.7 Робота з регіонами

Для рисування реальних предметів навколишнього світу недостатньо мати в арсеналі лише стандартні геометричні фігури. Необхідно також вміти їх комбінувати. Саме такі можливості і надають регіони.

Регіон – це графічний об'єкт Windows, в якому зберігається опис деякої області на екрані. Ця область може складатись не тільки із стандартних фігур, але також і з їх комбінацій. Над регіонами можна виконувати також багато інших операцій, зокрема:

- обводити границі регіонів, заповнювати або інвертувати зображення всередині них;
- перевіряти наявність в регіоні заданої точки, наприклад, позиції курсора миші;
- переміщати регіони;

Розглянемо спочатку, як створюються регіони з використанням відповідних API-функцій.

Створити регіон у вигляді прямокутника можна за допомогою API-функції *CreateRectRgn*():

HRGN CreateRectRgn(int nLeftRect, int nTopRect,
int nRightRect, int nBottomRect); .

Створити регіон у вигляді елліпса можна за допомогою API-функції *CreateEllipticRgn*():

HRGN CreateEllipticRgn(int nLeftRect, int nTopRect,
int nRightRect, int nBottomRect); .

Створити регіон у вигляді прямокутника із скругленими кутами можна за допомогою API-функції *CreateRoundRectRgn*():

HRGN CreateRoundRectRgn(int nLeftRect, int nTopRect,
int nRightRect, int nBooottomRect,
int nWidthEllipset, int nHeightEllipset); .

Створити регіон у вигляді багатокутника можна за допомогою API-функції *CreatePolygonRgn*():

*HRGN CreatePolygoncRgn(CONST POINT *lppt, int cPoints,*
int fnPolyFillMode); ,

де *lppt* представляють собою масив структур типу *POINT*, який містить координати набору точок, а *cPoints* – кількість самих точок. Параметр *fnPolyFillMode* вказує режим заповнення багатокутника: *ALTERNATE* (за замовчуванням), або *WINDING*. Значення цих режимів такі ж, як і для API-функції *SetPolyFillMode*().

Всі перераховані вище API-функції для роботи з регіонами повертають дескриптор регіону, який зберігається у змінній типу *HRGN*.

Після створення регіонів у вигляді стандартних фігур їх можна далі комбінувати у більш складні регіони за допомогою API-функції *CombineRgn*():

int CombineRgn(HRGN hrgnDest, HRGN hrgnSrc1, HRGN hrgnSrc2,
int fnCombineMode);

де *hrgnSrc1*, *hrgnSrc2* - дескриптори двох початкових регіонів, які потрібно скомбінувати, *hrgnDest* – дескриптор результуючого регіону, *fnCombineMode* - спосіб комбінування двох регіонів:

RGN_AND – результуючим регіоном буде спільна частина регіонів *hrgnSrc1* і *hrgnSrc2*;

RGN_OR – результуючим регіоном будуть всі частини регіонів *hrgnSrc1* і *hrgnSrc2*;

RGN_COPY – результуючим регіоном буде регіон *hrgnSrc1*.

RGN_DIFF – результуючим регіоном буде частина регіону *hrgnSrc1*, яка не збігається з регіоном *hrgnSrc2*.

RGN_XOR – результуючим регіоном будуть частина регіону *hrgnSrc1* і частина регіону *hrgnSrc2*, які не збігаються.

В API-функції *CombineRgn*() можна вказувати один і той же дескриптор для *hrgnDest* і *hrgnSrc1* або для *hrgnDest* і *hrgnSrc2*.

Після створення регіону його можна зафарбувати двома способами. Можна заповнити весь простір регіону, використовуючи вибраний в даний контекст пристрою пензель, за допомогою API-функції *PaintRgn*():

BOOL PaintRgn(HDC hdc, HRGN hrgn, HBRUSH hbr); .

Можна також заповнити весь простір регіону, за допомогою API-функції *FillRgn*(), якщо вказати для неї потрібний пензель:

BOOL FillRgn(HDC hdc, HRGN hrgn, HBRUSH hbr); .

Можна нарисувати границю навколо регіонів, використовуючи вибраний пензель, якщо скористатись API-функцією *FrameRgn*():

BOOL FrameRgn(HDC hdc, HRGN hrgn, HBRUSH hbr,
int nWidthEllipset, int nHeightEllipset); .

Для створення динамічних рисунків, де відбувається переміщення регіонів, дуже корисна API-функція *OffsetRgn*():

int OffsetRgn(HRGN hrgn, int nXOffset, int Yoffset); .

де *nXOffset* – відстань, на яку має бути переміщений регіон в горизонтальному напрямі; *nYOffset* – відстань, на яку має бути переміщено регіон у вертикальному напрямі;

За допомогою API-функції *PtInRegion*() можна перевірити попадання точки в регіон:

BOOL PtInRegion(HRGN hrgn, int X, int Y); .

Ця функція повертає *TRUE*, якщо точка з координатами *X* та *Y* знаходиться в регіоні, заданого дескриптором *hrgn*.

Після закінчення роботи з регіоном необхідно передати його дескриптор API-функції *DeleteObject*() для звільнення займаних ним ресурсів:

BOOL DeleteObject(HGDIOBJ hObject); .

Як приклад, розглянемо віконну функцію, в якій використовуються регіони. При натисканні правої клавіші миші в робочій області вікна програми з'являється регіон червоного кольору, отриманий комбінуванням регіонів, які задані API-функціями *CreateRectRgn*() та *CreateEllipticRgn*(). При натисканні лівої клавіші миші з'являється регіон синього кольору, отриманий комбінуванням регіонів, які задані API-функціями *CreatePolygonRgn*() та *CreateEllipticRg*(). Перед кожним виведенням вказаних даних робоча область вікна очищається білим кольором.

```
LRESULT CALLBACK WindowFunc(HWND hWnd,UINT iMessage,
                            WPARAM wParam,LPARAM lParam)
{
    HRGN Region1,Region2;
    HDC hdc;
    HBRUSH hbrushRed,hbrushBlue,hbrushWhite;
    POINT Points[3]= {{ 200,250},
                      { 300,50},
                      { 400,250}};
    switch(iMessage)
    {
     case WM_RBUTTONDOWN:
```

```
            hdc=GetDC(hWnd);
        hbrushWhite=CreateSolidBrush(RGB(255,255,255));
            SelectObject(hdc,hbrushWhite);
            Rectangle(hdc,0,0,600,400);
            Region1=CreateRectRgn(100,50,150,350);
            Region2=CreateEllipticRgn(50,100,200,200);
            CombineRgn(Region1,Region1,Region2,RGN_XOR);
            hbrushBlue=CreateSolidBrush(RGB(0,0,255));
            SelectObject(hdc,hbrushBlue);
            FillRgn(hdc,Region1,hbrushBlue);
            DeleteObject(Region1);
            DeleteObject(Region2);
         ReleaseDC(hWnd,hdc);
            break;
    case WM_LBUTTONDOWN:
            hdc=GetDC(hWnd);
       hbrushWhite=CreateSolidBrush(RGB(255,255,255));
            SelectObject(hdc,hbrushWhite);
            Rectangle(hdc,0,0,600,400);
            Region1=CreatePolygonRgn(Points,3,ALTERNATE);
            Region2=CreateEllipticRgn(250,200,350,300);
            CombineRgn(Region1,Region1,Region2,RGN_DIFF);
            hbrushRed=CreateSolidBrush(RGB(255,0,0));
            SelectObject(hdc,hbrushRed);
            PaintRgn(hdc,Region1);
            DeleteObject(Region1);
            DeleteObject(Region2);
            ReleaseDC(hWnd,hdc);
            break;
        case WM_DESTROY:
         PostQuitMessage(0); break;
        default:
        return(DefWindowProc(hWnd,iMessage,wParam,lParam));
        }
            return 0L;
}
```

2.3 Виведення тексту в API-програмах

На відміну від операційної системи *MS-DOS*, в системі *Windows* відсутній спеціальний текстовий режим, і тому, будь-яка видима інформація повинна бути оформлена графічними засобами. Розглянемо особливості виведення тексту в графічному середовищі *Windows*.

Як і при рисуванні геометричних фігур, використовується контекст пристрою і логічна система координат з початком в лівому верхньому кутку робочої області вікна.

Для задання кольору тексту використовується API-функція *SetTextColor*():

COLORREF SetTextColor(HDC hdc, COLORREF color); .

Колір фону може бути встановлений за допомогою API-функції *SetBkColor*(), формат якої був раніше розглянутий.

Способом відображення фону при виведенні тексту можна керувати за допомогою API-функції *SetBkMode*():

int SetBkMode(HDC hdc, int mode); .

Остання функція визначає режим *mode* відображення при виведенні фону тексту і деяких інших видів графіки. Параметр *mode* може задаватися одним із двох макросів. Якщо значення *mode* дорівнює *OPAQUE*, тоді колір фону при виведенні тексту кожен раз буде змінюватися на поточний колір фону, який задається функцією *SetBkColor*(). Якщо ж *mode* дорівнює *TRANSPARENT*, тоді колір фону при виведенні тексту не змінюється, і використання функції *SetBkColor*() не має сенсу. За замовчуванням значення *mode* дорівнює *OPAQUE*.

Дуже легко вивести один текстовий рядок за допомогою API-функції *TextOut*():

BOOL TextOut(HDC hdc, int X, int Y, LPSTR str, int length); .

В результаті рядок *str* довжиною *length* виводиться в робочу область вікна, починаючи з точки з координатами X і Y, (за замовчуванням ці координати задаються в пікселях) і відображаються на екрані чорним кольором тексту на поточному фоні вікна:

TextOut(hdc,X,Y,str,5); .

Якщо в самій API-функції *TextOut*() заданий текст, що виводиться, тоді параметр *length* можна не вказувати:

TextOut(hdc,X,Y,"word"); .

В протилежному випадку, коли замість текстового рядка вказано його ідентифікатор *str*, необхідно або явно вказати довжину рядка, або використати відому функцію *strlen*() для визначення його довжини:

TextOut(hdc,X,Y,str,strlen(str)); .

Для тексту, що виводиться можна також використовувати різні режими вирівнювання з допомогою API-функції *SetTextAlign*():

UINT SetTextAlign(HDC hdc, UINT mode); .

Параметр *mode* визначає взаємозв'язок між опорною точкою і прямокутником, який обмежує текст, і може приймати такі значення:

TA_LEFT – опорна точка розміщується на лівій межі бмежувального прямокутника;

TA_RIGHT – опорна точка розміщується на правій межі обмежувального прямокутника;

TA_BATTOM – опорна точка розміщується на нижній межі обмежувального прямокутника;

TA_TOP – опорна точка розміщується на верхній межі обмежувального прямокутника;

TA_BASELINE – опорна точка розміщується на базовій лінії тексту;

TA_CENTR – опорна точка вирівнюється по горизонталі відносно центру обмежувального прямокутника;

TA_NOUPDATECT – поточна позиція не змінюється після виведення тексту;

TA_UPDATED – поточна позиція змінюється після кожного виведення тексту;

За замовчуванням використовуються такі значення: *TA_LEFT, TA_TOP, TA_NOUPDATECT.*

Дуже важливою перевагою середовища *Windows* є те, що більшу частину роботи по виведенню інформації воно бере на себе. Багато параметрів, про існування яких програміст може і не здогадуватися, задаються за замовчуванням. Наприклад, при виведенні текстового рядка таких параметрів налічується біля двох десятків.

Однак, на жаль, багато роботи залишається і за програмістом, особливо при виведенні великих об'ємів текстової інформації. API-функція *TextOut*() ніяк не форматує текст, що виводиться, і навіть не виконує операцію повернення каретки і переходу на новий рядок. Багато проблем виникає і з шрифтами. Так що завдання виведення тексту в *Windows* не легше, аніж рисування графічних зображень.

Дуже корисно познайомитися із структурою *TEXTMETRIC*, яка містить метрики (характеристики) шрифту, що використовується:

```
typedef struct tag TEXTMETRIC
{
LONG tmHeight;          /*  повна висота шрифту                 */
LONG tmAscentt;         /*  висота над основною лінією          */
LONG tmDescent;         /*  розмір нижнього виступу             */
LONG tmInternalLeading; /*  розмір верхнього виступу            */
LONG tmExternalLeading; /*  міжрядковий інтервал                */
LONG tmAveCharWidth;    /*  середня ширина символу              */
LONG tmMaxCharWidth;    /*  максимальна ширина символу          */
LONG tmWeidth;          /*  насиченість шрифту                  */
LONG tmOverhang;        /*  додаткова ширина символу            */
LONG tmDigitizedAspectX; /*  горизонтальний аспект              */
LONG tmDigitizedAspect; /*  вертикальний аспект                 */
BYTE tmFirstChar;       /*  перший символ                       */
BYTE tmLastChar;        /*  останній символ                     */
BYTE tmDefaultChar;     /*  символ за замовчуванням             */
BYTE tmBreakChar;       /*  символ для позначення межі слова    */
BYTE tmItalic;          /*  не 0, якщо шрифт нахилений          */
```

```
BYTE tmUnderlined;      /* не 0, якщо букви підкреслені      */
BYTE tmStructOut;       /* не 0, якщо букви закреслені       */
BYTE tmPitchAndFamily;  /* сімейство і гарнітура             */
BYTE tmCharSet;         /* ідентифікатор множини символів    */
};.
```

Будь–який із перерахованих вище параметрів структури *TEXTMETRIC* можна отримати за допомогою API-функції *GetTextMetric*():

BOOL GetTextMetric(HDC hdc, LPTEXTMETRIC lpTAttrib); .

Другий параметр функції і є покажчиком розглянутої структури.

Оскільки різні символи одного і того ж самого шрифту можуть мати різну ширину, тому для точного визначення довжини рядка потрібно скористатись API-функцією *GetTexExtenPoint32*():

BOOL GetTexExtenPoint32(HDC hdc, LPSTR str, int len, LPSTR size); .

Тут параметр *len* задає кількість символів досліджуваного рядка *str*. Обчислена довжина і висота текстових рядків в логічних одиницях записуються в структуру *SIZE*, покажчик на яку задається параметром *size*. Сама структура *SIZE* визначається таким чином:

```
typedef struct tagSIZE
{
LONG cx;            /* довжина рядка     */
LONG cy;            /* висота рядка      */
}     SIZE;.
```

Дуже корисною є API-функція *GetSystemMetric*(), яка дозволяє визначити системні метрики:

int GetSystemMetric(int what); .

Параметр *what* визначає шукану величину, наприклад:

SM_CXFULLSCREEN – ширина робочої області максимізованого вікна;

SM_CYFULLSCREEN – висота робочої області максимізованого вікна;

SM_CXSCREEN – ширина екрана;

SM_CYSCREEN – висота екрана.

Як приклад, розглянемо віконну функцію, в якій програмується виведення чотирьох текстових рядків:

Результат
Виконання програми:
a=5 b=3 c=8
Розміри екрана ...

Координати початку першого рядка: *X=50, Y=20*. Координата *Y* кожного наступного рядка визначається за допомогою метрик *tmHeight* і *tmExternalLeading* структури *TEXTMETRIC*.

```
LRESULT CALLBACK WindowFunc(HWND hWnd,UINT iMessage,
                            WPARAM wParam,LPARAM lParam)
{
      HDC hdc;   TEXTMETRIC tm;
      PAINTSTRUCT ps;
      char str[80];
      int a,b,c,X,Y,maxX,maxY;
      a=5; b=3;     c=a+b;
      switch(iMessage)
      {
 case WM_CREATE:
            maxX=GetSystemMetrics(SM_CXSCREEN);
            maxY=GetSystemMetrics(SM_CYSCREEN);
           break;
  case WM_PAINT:
            hdc=BeginPaint(hWnd,&ps);
            SetTextColor(hdc,RGB(255,0,0));
            SetBkColor(hdc,RGB(0,255,255));
            sprintf(str,"Результамт");
                  X=50;  Y=20;
            TextOut(hdc,X,Y,str,strlen(str));
                  Y=Y+tm.tmHeight+tm.tmExternalLeading;
           /*  next   string       */
            strcpy(str,"виконання програми:");
            TextOut(hdc,X,Y,str,strlen(str));
                  Y=Y+tm.tmHeight+tm.tmExternalLeading;
            /*   next string          */
            sprintf(str,"a=%d  b=%d  c=%d",a,b,c);
           TextOut(hdc,X,Y,str,strlen(str));
                  Y=Y+tm.tmHeight+tm.tmExternalLeading;
           /*   next string          */
            sprintf(str,"Розміри екрану %d на %d",maxX,maxY);
            TextOut(hdc,X,Y,str,strlen(str));
           EndPaint(hWnd,&ps);
           break;
      case WM_DESTROY:
            PostQuitMessage(0); break;
                  default:
       return(DefWindowProc(hWnd,iMessage,wParam,lParam));
     }
         return 0L;
}
```

2.4 Розробка меню в API-програмах

Меню є найбільш поширеним елементом інтерфейсу в середовищі *Windows*. Всі головні вікна системних та прикладних програм мають меню. Воно, за звичай, міститься в верхній частині вікна і дозволяє працювати з командами і модифікаторами. Команди в меню означають визначені дії (наприклад, виклик функції), а модифікатори - деякі особливості меню:

√ (маркер) - означає вибір даного пункту меню;

F (знак підкреслення) - означає можливість вибору даного пункту меню за допомогою однієї клавіші на клавіатурі (акселератора);

→ (стрілка) - означає появу висхідного меню;

... (крапки) - означає появу діалогового вікна після вибору даного пункту меню;

- напівтоновий колір меню - означає його недоступність;

— (лінія розділу) - відділяє одну групу елементів від іншої.

Особливим варіантом меню є контекстне або висхідне меню. Воно може з'явитися в будь-якому місці екрана після натиснення правої кнопки миші.

Стандартний спосіб створення меню в програмі складається із таких етапів:

1. Створення шаблону меню;
2. Підключення меню до програми;
3. Обробка повідомлень меню.

Шаблон меню створюється в двох файлах: основному файлі ресурсів (типу *.rc) і файлі ідентифікаторів ресурсів Resourse.h.

В найпростішому випадку файл ресурсів може бути підготовлений в будь-якому текстовому редакторі і мати, наприклад, такий вигляд.

```
MenuName MENU [параметри]
{
   елементи меню
}.
```

Тут *MenuName* - ім'я меню будь-якого виду. Як необов'язкові, можуть бути використані такі параметри меню:

DISCARDABLE - невикористовуване меню може бути вилучене з пам'яті.

FIXED - меню зафіксоване в пам'яті.

LOADONCALL - меню завантажується тільки перед його безпосереднім використанням.

MOVEABLE - меню може переміщуватися в пам'яті.

PRELOAD - меню завантажується в момент запуску програми.

Існує два типи елементів меню: MENUITEM і POPUP. Перший оператор визначає звичайний елемент меню і має формат:

MENUITEM "Ім'я", MenuID [, параметри].

Другий оператор визначає низхідне підменю і має формат:

POPUP "Ім'я" [, параметри].

В обох форматах *Ім'я* - будь-яке ім'я пункту меню, а в якості необов'язкових параметрів найчастіше використовуються такі:

CHECKED - біля імені пункту меню відображається маркер;

CREYED - пункт меню недоступний і відображається напівтоновим кольором;

INACTIVE - пункт меню заблокований і відображається чорним кольором;

SEPARATOR - горизонтальна лінія розділу.

Важливу роль відіграє ідентифікатор пункту меню *Menu ID*. Кожен пункт меню повинен мати унікальний цілочисельний (від 1 до 65000) ідентифікатор; саме цей ідентифікатор служить зв'язувальною ланкою між шаблоном меню і програмою (файлом *.cpp). Програмісту незручно працювати з цілочисельними ідентифікаторами, тому їм дають зазвичай змістовні імена, а відповідність між такими іменем і числом задається спеціальною таблицею відповідностей. Звідси випливають три варіанти використання ідентифікаторів:

а) в операторі *MENUITEM* як ідентифікатор записати безпосередньо число;

б) в операторі *MENUITEM* як ідентифікатор записати змістовне ім`я, а відповідність між іменем і числом визначити в спеціальній таблиці;

в) відрізняється від попереднього варіанту тим, що таблиця відповідностей поміщається в окремий файл *Resource.h*.

На практиці частіше використовується третій варіант. Ми спеціально детально зупинились на цьому питанні, оскільки аналогічний принцип використовується також і при утворенні інших типів ресурсів (діалогу і т.д.).

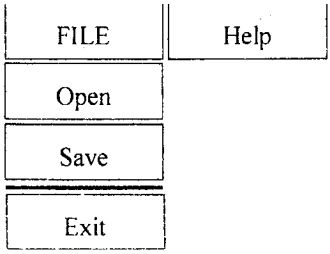


Рисунок 2.7 - Приклад меню

Наведемо опис меню, який відповідає рис.2.7.

```
IDR_MENU1 MENU DISCARDABLE
{
    POPUP "&File"
      {
            MENUITEM "&Open", IDM_OPEN
            MENUITEM "&Save", IDM_SAVE
            MENUITEM SEPARATOR
            MENUITEM "&Exit", IDM_EXIT
      }
            MENUITEM "&Help", IDM_HELP
}
#define     IDR_MENU1     100
#define     IDM_OPEN      101
#define     IDM_SAVE      102
#define     IDM_EXIT      103
#define     IDM_HELP      104
```

Таким чином, шаблон меню можна створити чисто програмним способом. Однак, пакет Visual C++ надає можливість також і візуального проектування меню за допомогою спеціального редактора ресурсів. Для цього необхідно створити спочатку порожній файл ресурсів (типу *.rc), а потім вставити в нього ресурс меню. Детальніше процес візуального створення нового ресурсу розглядається в розділі 1. Результатом роботи редактора ресурсу є ті ж файли, які можна створити програмним шляхом.

Після створення меню (програмно або візуально) його необхідно підключити до основної програми таким чином.

1. На початку основного файлу (типу *. cpp) написати :

#include "Resource.h" .

2. У файлі *Resource.h* дізнатись про ідентифікатор ресурсу меню (наприклад, *IDR_MENU1*) і вказати його у визначенні класу вікна:

WndClass.lpszMenuName=(LPSTR)IDR_MENU1; .

Якщо запустити проект на виконання, тоді в головному вікні програми ми вже зможемо побачити створене меню. Зрозуміло, що ніякої реакції на вибір пунктів меню не буде. Необхідно ще записати в віконній функції обробку повідомлень меню. Простим способом такої обробки повідомлень може бути поява заданого вікна повідомлень на вибір відповідного пункту меню. Наприклад, обробка для пункту меню *Open*:

```
case IDM_OPEN:
MessageBox(hWnd, "Вибрано Open", "Open", MB_OK)
        break; .
```

2.5 Програмування діалогу в API-програмах

Діалог є спеціальним типом вікна, за допомогою якого користувач може взаємодіяти з програмою.

Діалог взаємодіє з користувачем за допомогою одного або декількох елементів керування. Елемент керування (ЕК) – це особливий вид вікна, який призначений для виведення або введення інформації.

Основні типи ЕК:

- кнопка (*BUTTON, PUSHBUTTON*);
- перемикач (*CHECKBOX*);
- селекторна кнопка (*RADIOBATTON*);
- вікно введення (*EDIT*);
- список (*LISTBOX*);
- комбінований список (*COMBOBOX*);
- статичний елемент (*STATIC*);
- лінійка прокручування (*SCROOLLBAR*).

ЕК можуть генерувати повідомлення і приймати їх. Повідомлення, які генерує ЕК, ініціюють дії користувача. Повідомлення для ЕК представляють собою команди, на які ці елементи повинні реагувати.

Діалоги бувають двох типів:

- модальні;
- немодальні.

Модальні вікна діалогу блокують всі інші вікна прикладних програм так, що нічого не можна зробити, поки вони не будуть закриті.

Немодальні діалог не затримує виконання програми, дозволяється переключення між діалогом і іншими вікнами прикладної програми. Створення і наявність блоку діалогу потребує одночасної наявності трьох компонентів:

- шаблону блока діалогу;
- програмного коду, який утворює і відображає його на екрані;
- діалогової процедури, яка обслуговує взаємодію користувача з блоком діалогу.

2.5.1 Створення шаблону блока діалогу

Шаблон блока діалогу, описує форму і розміщення самого діалогового вікна і всіх його внутрішніх ЕК на екрані. Подібно меню, його можна створити програмно, записавши відповідні оператори в ресурсні файли (файл типу **.rc* і файл *Resourse.h*).

Однак набагато краще візуально створити діалогове вікно, і тоді система автоматично згенерує потрібні ресурсні файли. Саме в цьому і полягає суть візуального програмування.

Для того, щоб добавити діалог в проект, в якому відсутні інші ресурси, необхідно спочатку створити пусті ресурсні файли таким же чином, як це було показано в першому розділі. А якщо такі файли вже існуть, тоді залишається лише виконати такі команди.

В головному меню пакета Visual C++ необхідно вибрати команди *Insert/Resource*, далі вибрати рядок "*Dialog*", і натиснути кнопку "*New*". Після цього починає працювати редактор діалогових вікон.

Спочатку редактор надає діалогове вікно, яке містить лише дві кнопки: *Ok* і *Cancel*. Поруч знаходиться панель інструментів із стандартними ЕК (при відсутності панелі інструментів її можна легко викликати через команди головного меню *Tools/Customize/Toolbars*).

Далі задані ЕК із панелі інструментів за допомогою миші необхідно перенести на діалогове вікно. Приблизно розташувати ЕК можна за допомогою миші, а для точного переміщення ЕК рекомендується скористатись командами меню *Layout* (Розміщення) або кнопкою панелі інструментів *Dialog*.

Потім починається етап настроювання параметрів кожного ЕК. Для цього достатньо викликати команду *Properties* (Властивості) із контекстного меню (викликається правою клавішею миші) відповідного ЕК і у вікні настроювання, яке відкривається, задати деякі параметри: підписи, початковий стан ЕК тощо. Більшість параметрів настроювання редактор встановлює самостійно і з ними можна погодитись. Таким же чином можна викликати контекстне меню всього діалогового вікна і теж задати додаткові параметри: назву вікна, колір фону, додаткові кнопки керування та ін.

Із всіх параметрів настроювання найважливішими є ідентифікатори діалогового вікна та кожного ЕК. Саме вони будуть надалі зустрічатись в інших файлах проекту, тому їх необхідно добре запам'ятати.

Після закінчення формування діалогового вікна можна переглянути його закінчений вигляд, якщо викликати із головного меню команду *Layout/Test*.

Корисно переглянути ресурсний файл і проаналізувати створений редактором шаблон блока діалогу.

Діалогове вікно описується в шаблоні блока діалогу оператором *DIALOG*, який має такий формат:

IDD_name DIALOG <liding><memory>, X, Y, Width, Height, <style>,
{
елементи діалогу
},

де *IDD_name* – ідентифікатор діалогового вікна;
<liding> - спосіб завантаження діалогового вікна;
<memory> - режим роботи з пам'яттю;
X, Y – координати лівого верхнього кута діалогового вікна;
Width, Height – ширина і висота діалогового вікна;

<style> - стилі діалогового вікна:

DS_MODALFRAME – утворення модального діалогу;

WS_CAPTION – утворення вікна із заголовком;

WS_SYSMENU – утворення вікна із системним меню;

WS_POPUP – утворення висхідного вікна;

WS_MINIMIZEBOX – утворення вікна із кнопкою мінімізації;

WS_VISIBLE – відображення вікна при активізації.

Елементами діалогового вікна є ЕК. Кожний ЕК описується в шаблоні блока діалогу одним оператором, формат якого такий:

<тип ЕК> <текст>, ID, X, Y, Width, Height, <style>.

де *<тип ЕК>* - назва конкретного ЕК, наприклад, *LISTBOX*;

<текст> - текстовий рядок, асоційований із ЕК.

Інші параметри мають ті ж значення, що і відповідні параметри оператора *DIALOG.*

Замість конкретних операторів можна використовувати універсальний оператор *CONTROL*, придатний для будь-якого ЕК. Цей оператор має такий формат:

CONTROL <текст>, ID, <клас>, <стилі>, X, Y, Width, Height: .

де *<клас>*- імя віконного класу (*BUTTON, COMBOBOX, EDIT, LISTBOX, SCROLLBAR, STATIC*), за допомогою якого можна створити конкретний ЕК.

Інші параметри оператора *CONTROL* мають ті ж значення, що і відповідні параметри оператора *DIALOG.*

Як приклад розглянемо шаблон блока діалогу, який містить, окрім кнопок *Ok* і *Cancel* (ідентифікатори відповідно *IDOK* і *IDCANCEL*), два перемикачі (*IDC_CHECK1, IDC_CHECK2*), дві селекторні кнопки (*IDC_RADIO1, IDC_RADIO2*), вікно введення (*IDC_EDIT1*) і список (*LISTBOX*).

```
IDD_DIALOG DIALOG DISCARDABLE  0, 0, 220, 120
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Діалог"
FONT 12, "MS Sans Serif"
BEGIN
    PUSHBUTTON    "OK", IDOK,7,74,40,14, BS_CENTER
    PUSHBUTTON    "Cancel", IDCANCEL, 171,73,40,14, BS_CENTER
    LISTBOX          IDC_LIST1, 156,6,48,40, LBS_SORT |
    LBS_NOINTEGRALHEIGHT | WS_VSCROLL | WS_TABSTOP
    CONTROL       "Check1", IDC_CHECK1,"Button",BS_AUTOCHECKBOX
                  WS_TABSTOP, 83,77,40,10
    CONTROL       "Check2", IDC_CHECK2,"Button",BS_AUTOCHECKBOX
                  WS_TABSTOP, 123,77,40,10
    EDITTEXT      IDC_EDIT1, 88,25,40,14, ES_AUTOHSCROLL
```

```
    CONTROL      "Radio1", IDC_RADIO1, "Button",
                 BS_AUTORADIOBUTTON, 82,57,39,10
    CONTROL      "Radio2", IDC_RADIO2, "Button",
                 BS_AUTORADIOBUTTON, 124,56, 39,10
END.
```

2.5.2 Створення і відображення діалогового вікна на екрані

Для створення модального діалогу використовується API-функція *DialogBox*():

int DialogBox(hInstance, lpName, hWnd, lpDFunc); ,

а для створення немодального діалогу використовується API-функція *CreateDialog*():

CreateDialog(hInstance, lpName, hWnd, lpDFunc); ,

де *hInstance* – дескриптор поточної прикладної програми, який передається в функцію *WinMain()* як параметр;

lpName – ім'я ресурсу діалогового вікна; часто використовується макрос *MAKEINTRESOURSE*, що перетворює числовий ідентифікатор ресурсу в рядок тексту.

hWnd – дескриптор вікна, який породжує діалог;

lpDFunc – покажчик на діалогову процедуру.

Як приклад, розглянемо віконну функцію, в якій викликається API-функція *DialogBox*().

```
LRESULT CALLBACK WindowFunc(HWND hWnd,UINT iMessage,
                            WPARAM wParam,LPARAM lParam)
{
          HDC hdc;
  HPEN hRedPen;
 HBRUSH hbrushYellow,hbrushBlue;
     switch(iMessage)
     {
        case WM_COMMAND:
    switch(wParam)
    {
     case IDM_STATUS:
                  if ((status1==1)&&(status2==0))
                  {
              hdc=GetDC(hWnd);
              hRedPen=CreatePen(PS_SOLID,4,RGB(255,0,0));
              SelectObject(hdc,hRedPen);
              hbrushYellow=CreateSolidBrush(RGB(255,255,0));
              SelectObject(hdc,hbrushYellow);
```

```
            Ellipse(hdc,150,100,250,200);
            hbrushBlue=CreateSolidBrush(RGB(0,0,255));
            SelectObject(hdc,hbrushBlue);
            Rectangle(hdc,300,200,500,300);
                }
                        break;
    case IDM_DIALOG:
DialogBox(hInst,MAKEINTRESOURCE(IDD_DIALOG),hWnd,DialogFunc);
break;
            case IDM_QUIT:
                PostQuitMessage(0);          break;
  }   break;
     case WM_DESTROY:
      PostQuitMessage(0); break;
                default:
    return(DefWindowProc(hWnd,iMessage,wParam,lParam));
  }
      return 0L;
}
```

2.5.3 Створення діалогової процедури

Призначення діалогової процедури – обробка повідомлень, пов'язаних з обміном даних у діалоговому вікні. Діалогова процедура нагадує віконну функцію, однак має ряд суттєвих відмінностей від останньої. Найголовніша відмінність полягає в тому, що діалогова процедура обробляє повідомлення тільки від самого блока діалогу або від ЕК і не передає неупізнані повідомлення API-функції *DefWindowProc*().

Назвемо діалогову процедуру *DialogFunc*(). Формат її заголовка такий:

```
BOOL CALLBACK DialogFunc(HWND hdWnd,UINT iMessage,
                         WPARAM wParam, LPARAM lParam);
```

Діалоговій процедурі передаються ті ж чотири параметри, що і віконній функції, однак перший параметр *hdWnd* означає тут дескриптор діалогового вікна.

При першому відображенні діалогового вікна його діалогова процедура отримує перше повідомлення *WM_INITDIALOG*. Обробка цього повідомлення полягає в заданні початкових значень для ЕК діалогового вікна. Це повідомлення можна вважати аналогом повідомлення *WM_CREATE* для віконної функції. Точно так, аналогом повідомлення *WM_DESTROY* є виклик API-функції *EndDialog*() для закінчення роботи з діалоговим вікном.

Між ініціалізацією і закриттям діалогового вікна відбувається обробка повідомлень, які посилаються ЕК та отримуються від них.

Кожного разу при активізації ЕК у діалогову процедуру надходить повідомлення *WM_COMMAND*, в якому параметр *lParam* містить ідентифікатор цього ЕК, а параметр *wParam* може містити додаткову інформацію.

Для обміну даними з конкретними ЕК існують свої API-функції. Найбільш універсальною API-функцією для посилання повідомлень будь-якому ЕК є API-функція *SendDlgItemMessage*():

LONG SendDlgItemMessage(HWND hdWnd, int ID, UINT IDMsg, WPARAM wParam, LPARAM lParam); .

Ця функція направляє повідомлення, що міститься в параметрі *IDMsg*, елементу керування з ідентифікатором *ID*. Інші параметри мають той же смисл, що і для діалогової процедури.

Найбільш універсальною API-функцією для отримання повідомлень від будь-якого ЕК є API-функція *GetDlgItemMessage*():

HWND GetDlgItemMessage(HWND hdWnd, int ID); .

Використовуючи останні дві API-функції можна працювати з будь-яким ЕК. Поруч з ними існують також і спеціальні API-функції для конкретних ЕК. Розглянемо коротко особливості обміну даними з найбільш поширеними ЕК.

При роботі зі списками застосовуються дві основні операції:

- ініціалізація списку;
- визначення реакції на вибір елемента керування.

Ініціалізація списку здійснюється при його першому відображенні шляхом обробки повідомлення *WM_INITDIALOG* в діалоговій процедурі *DialogFunc()*:

```
case WM_INITDIALOG:
SendDlgItemMessage(hdWnd, ID, LB1, LB_ADDSTRING, 0,
(LPARAM)str);
return 1;
```

Після ініціалізації список готовий до роботи з ним. Існує два способи вибору елементів керування списком:

1. Подвійне натискання миші на елементі, що вибирається. В цьому випадку програма повинна обробляти вибір відразу. Діалоговій процедурі *DialogFunc*() буде направлено повідомлення *WM_COMMAND*, в якому *LOWORD(wParam)* буде містити ідентифікатор вікна списку, а *HIWORD(wParam)* – повідомлення *LBN_DBLCLK*.

2. Одинарне натискання миші або натискання клавіші клавіатури на вибраному елементі. В цьому випадку повідомлення в програму не надсилається, а лише запам'ятовується вибраний елемент списку. Коли виникає необхідність, тоді списку надсилається повідомлення *LB_GETCURSEL*. У відповідь список повертає номер вибраного елемента, або повертає *LB_ERR(-1)*, якщо ніякий елемент не вибрано. Наприклад, для вибору і-го елемента списку в діалоговій процедурі *DialogFunc*() можна записати:

i=SendDlgItemMessage(hdWnd,ID_LB1?LB_GETCURSEL, 0,0L);

Вікна введення застосовуються для введення текстових рядків. Подібно списку, вікно введення може не тільки приймати повідомлення (за допомогою API-функції *SendDlgItemMessage*()), але також і генерувати їх, наприклад, за допомогою API-функції *GetDlgItemText*():

GetDlgItemText(hdWnd, ID_EDIT1,str,80).

В результаті введений з клавіатури рядок довжиною до 80 символів буде міститись у змінній *str*.

Як приклад, розглянемо діалогову процедуру, в якій передбачені такі операції обміну із ЕК:

- початкова ініціалізація списку трьома текстовими рядками;
- початкова ініціалізація перемикачів нулями;
- початкова ініціалізація першої селекторної кнопки;
- виведення вікна повідомлення *MessageBox* із номером вибраного рядка списка при подвійному натисненні миші на цьому рядку;
- виведення вікна повідомлення *MessageBox* із текстовим рядком, який був набраний у вікні введення;
- отримання стану перемикачів у змінних *status1* та *status2*.

Отримані дані з діалогового вікна далі можуть передаватись у віконну функцію, яка викликала діалогову процедуру. В нашому прикладі наведена раніше віконна функція викликає діалогову процедуру, що розглядається далі. Якщо був вибраний тільки перший перемикач (тобто *status=1, status2=0*), тоді після закриття діалогового вікна і виклику пункту меню на екрані з'являється рисунок. Змінні *status1* та *status2* оголошені на початку програми, як глобальні:

int status1, status2;

```
BOOL CALLBACK DialogFunc(HWND hdWnd, UINT message,
                         WPARAM wParam, LPARAM lParam)
{
    int i;  char buf[80],str[80];
  switch(message)
  {
        case WM_INITDIALOG:
            SendDlgItemMessage(hdWnd,IDC_CHECK1,
                         BM_SETCHECK,0,0L);
            SendDlgItemMessage(hdWnd,IDC_CHECK2,
                         BM_SETCHECK,0,0L);
            SendDlgItemMessage(hdWnd,IDC_RADIO1,
                         BM_SETCHECK,1,0L);
            SendDlgItemMessage(hdWnd,IDC_RADIO2,
                         BM_SETCHECK,0,0L);
            SendDlgItemMessage(hdWnd,IDC_LIST1,
```

```
                    LB_ADDSTRING,0,(LONG)"String1");
                SendDlgItemMessage(hdWnd,IDC_LIST1,
                    LB_ADDSTRING,0,(LONG)"String2");
                SendDlgItemMessage(hdWnd,IDC_LIST1,
                    LB_ADDSTRING,0,(LONG)"String3");
        return TRUE;
          case WM_COMMAND:
    switch(LOWORD(wParam))
          {
              case IDC_LIST1:
                  if(HIWORD(wParam) == LBN_DBLCLK)
                  {
                      i=SendDlgItemMessage(hdWnd,IDC_LIST1,
                          LB_GETCURSEL,0,0L);
                    sprintf(buf,"%d",i+1);
                  MessageBox(hdWnd,buf,"Button1",MB_OK);
                  }
                   return TRUE;
              case IDOK:
                  status1=SendDlgItemMessage(hdWnd,IDC_CHECK1,
                          BM_GETCHECK,0,0);
                  status2=SendDlgItemMessage(hdWnd,IDC_CHECK2,
                          BM_GETCHECK, 0,0);
                  EndDialog(hdWnd,TRUE);
                  GetDlgItemText(hdWnd, IDC_EDIT1,str,sizeof(str));
                  MessageBox(hdWnd,str,"Ok",MB_OK);
                   return TRUE;
        case IDCANCEL:
      EndDialog(hdWnd,TRUE);
                return TRUE;
          }
    return TRUE;
      }
  return FALSE;
}
```

2.6 Керування процесами і потоками в API-програмах

Всі операційні системи фірми Microsoft, починаючи з Windows 95 і Windows NT, є багатозадачними, тобто дозволяють одночасно виконуватись кільком програмам. При цьому підтримуються два види багатозадачності: процесна і потокова.

Тому спочатку розглянемо детальніше поняття "процес" і "потік".

2.6.1 Процесна багатозадачність

Під процесом в операційних системах Windows розуміють програму, яка виконується. Кожний раз після запуску на виконання файлу типу EXE формується новий процес з окремим захищеним адресним простором розміром 4Гбайт і автоматично створюється один потік виконання програми. Не тільки для різних, але і для однієї і тієї ж програми, викликаної кілька разів створюється новий процес.

Процеси "бачуть" виділені їм пам'ять та інші ресурси, нічого не знаючи про існування інших процесів. Тим самим забезпечується не тільки паралельна робота кількох процесів, а також і їх взаємний захист.

Таким чином, кожний процес може вважати себе єдиною програмою, що виконується в системі. Завдання операційної системи якраз і полягає в тому, щоб на одному центральному процесорі забезпечити одночасне виконання кількох процесів.

Керування процесами з боку програміста полягає в організації створення нового процесу із заданими характеристиками. Іншими словами, в тілі однієї програми має міститись відповідний програмний код для запуску на виконання ще однієї програми та, при необхідності, її завершення. Для цього в програмах мовою С існує декілька варіантів.

Ще з часів операційної системи MS-DOS відома функція *system*():

*int system(char *com_line);* .

Ця функція дозволяє не тільки виконувати команди DOS, але і запускати на виконання програми. Наприклад, для запуску на виконання програми із файлу "c:\prog.exe", необхідно записати:

system("c:\prog.exe"); .

Хоч функція *system*() і має дуже простий формат, однак навіть в режимі MS-DOS ця функція має два недоліки: великі втрати часу та пам'яті через необхідність завантаження в оперативну пам'ять другої копії COMMAND.COM. Функція *system*() не дозволяє також використати всі можливості віконного інтерфейсу Windows.

В оболонці Windows 3.x була введена нова функція для керування процесами – функція *WinExec*(), формат якої такий:

UINT WinExec(LPCSTR lpCmdLine, UINT nCmdShow); .

Ім'я файла, з якого запускається програма на виконання, задається параметром *lpCmdLine*, а спосіб відображення головного вікна нового процесу – параметром *nCmdShow* .

Наприклад, для запуску на виконання програми із файлу "c:\prog.exe" і відображенням на екрані головного вікна програми, необхідно записати:

WinExec("c:\prog.exe",SW_SHOW); .

В Windows 95/98 функція *WinExec*() також підтримується, однак вважається вже застарілою.

Максимальні можливості з керування процесами в середовищі Windows 95/98 надає API-функція *CreateProcess*(), формат якої такий:

```
BOOL CreateProcess(LPCSTR lpszName, LPSTR lpszComLine,
                   LPSECURITY_ATTRIBUTES lpProcAttr,
                   LPSECURITY_ATTRIBUTES lpThreadAttr,
                   BOOL, DWORD How,
                   LPVOID lpEnv, LPSTR lpszDir,
                   LPROCESS_INFORMATION lpPInfo);
```

Пояснимо коротко кожен із параметрів цієї функції.

Параметр *lpszName* повинен містити ім'я файлу, з якого запускається програма на виконання. Аргументи командного рядка задаються у буфері, покажчик на який передається параметром *lpszComLine*. Однак, якщо значення *lpszName* задається як *NULL*, тоді перше слово у командному рядку *lpszComLine* інтерпретується як ім'я файла, з якого запускається програма на виконання.

Параметри *lpProcAttr* та *lpThreadAttr* використовуються для задання атрибутів доступу створюваного процесу. Якщо вони використовуватись не будуть, тоді вони записуються як *NULL*.

Параметр *InheritAttr* задає дескриптори створюваного процесу. Якщо значення цього параметра рівне TRUE, тоді дескриптори нового процесу будуть успадковувати дескриптори батьківського процесу, а якщо FALSE - тоді дескриптори нового процесу не успадковуються.

Зазвичай новий процес виконується "нормально", якщо параметр *How* заданий як *NULL*, хоча при необхідності можна задати додаткові атрибути.

Параметр *lpEnv* є покажчиком на буфер, який містить параметри середовища для створюваного процесу. Якщо він рівний *NULL*, тоді новий процес успадковує середовище батьківського процесу.

Параметр *lpszDir* задає пристрій і каталог для створюваного процесу. Якщо він рівний *NULL*, тоді новий процес успадковує пристрій і каталог батьківського процесу.

Параметр *lpPInfo* є покажчиком на структуру *STARTUPINFO*, що містить інформацію про вигляд головного вікна створюваного процесу. Ця структура визначається наступним чином:

```
typedef struct STARTUPINFO
{
    DWORD   cb;          /* розмір структури STARTUPINFO                */
    LPSTR   lpReserved;  /* має бути NULL                               */
    LPSTR   lpDesktop;   /* ім'я "робочого стола"                       */
    LPSTR   lpTitle;     /* заголовок консолі (тільки для консолі) */
    DWORD   dwX;         /* X лівого верхнього кута                     */
    DWORD   dwY;         /* Y лівого верхнього кута                     */
    DWORD   dwXSize;     /* ширина нового вікна                         */
    DWORD   dwYSize;     /* висота нового вікна                         */
```

```
    DWORD    dwXCountChars;  /*  розмір буфера консолі          */
    DWORD    dwYCountChars;  /*  розмір буфера консолі          */
    DWORD    dwFillAttribute;/*  початковий колір тексту        */
    DWORD    dwFlags;        /*  визначення активних полів      */
    WORD     wShowWindow;    /*  спосіб відображення вікна      */
    WORD     cbReserved2;    /*  має бути  0                    */
    LPBYTE   lpReserved2;    /*  має бути  NULL                 */
    HANDLE   hStdInput;      /*  стандартні дескриптори         */
    HANDLE   hStdOutput;     /*  стандартні дескриптори         */
    HANDLE   hStdError;      /*  стандартні дескриптори         */
}
    STARTUPINFO; .
```

Із перерахованих вище полів структури *STARTUPINFO* обов'язковим є лише задання поля *cb*, яка визначає розмір структури. Значення більшості інших полів приймаються за замовчуванням (як правило, мають значення - *NULL*). В такому випадку на екрані відразу відкривається головне вікно створеного процесу. При бажанні головне вікно можна також відразу розвернути на весь екран, якщо задати такі параметри:

StartupInfo.dwFlags=STARTF_USESHOWWINDOW;
StartupInfo.wShowWindow=SW_SHOWMAXIMIZED; .

або звернути у піктограму:

StartupInfo.dwFlags=STARTF_USESHOWWINDOW;
StartupInfo.wShowWindow=SW_SHOWMINIMIZED; .

Останнім параметром API-функції *CreateProcess*() є *lpPInfo* - покажчик на структуру *PROCESS_INFORMATION*, яка визначається таким чином:

```
Typedef struct  PROCESS_INFORMATION
{
    HANDLE hProcess;          /* дескриптор нового процесу      */
    HANDLE hThread;           /* дескриптор головного потоку    */
    DWORD dwProcessID;        /* ідентифікатор нового процесу   */
    DWORD dwThreadID;         /* ідентифікатор головного потоку*/

}
    PROCESS_INFORMATION; .
```

Хоч список параметрів API-функції *CreateProcess*() досить великий, однак більшість із них приймається за замовчуванням, так що практичне використання цієї функції нескладне.

Створений процес не залежить від батьківського, але останній може достроково ліквідувати новий процес за допомогою API-функції *TerminateProcess*():

TerminateProcess(HANDLE hProcess, UINT status); .

Параметр *hProcess* визначає дескриптор нового процесу, який можна отримати з поля *hProcess* інформаційної структури в API-функції

CreateProcess() під час створення процесу. Значення параметра *status* задає код повернення вилученого процесу.

API-функції *CreateProcess*() і *TerminateProcess* повертають ненульове значення при успішному завершенні, і нульове – в іншому випадку.

Наступна програма показує на прикладі створення нового процесу з запуском на виконання програми із файлу *"c:\\test.exe"*.

```
LRESULT CALLBACK WindowFunc(HWND hWnd,UINT iMessage,
                            WPARAM wParam,LPARAM lParam)
{
 STARTUPINFO StartupInfo;
 PROCESS_INFORMATION  ProcessInfo;
 HANDLE hProcess1;
 ProcessInfo.hProcess=hProcess1;
      switch(iMessage)
 {
  case WM_COMMAND:
   switch(LOWORD(wParam))
   {
    case IDM_PROCESS:
   memset(&StartupInfo,0,sizeof(StartupInfo));
       StartupInfo.cb=sizeof(STARTUPINFO);
       StartupInfo.dwFlags=STARTF_USESHOWWINDOW;
       StartupInfo.wShowWindow=SW_SHOWMINIMIZED;
   CreateProcess(NULL,"c:\\test.exe",
         NULL,NULL,FALSE,0,NULL,NULL,
         &StartupInfo,&ProcessInfo);
   }    break;
      case WM_DESTROY:
       PostQuitMessage(0); break;
                default:
     return(DefWindowProc(hWnd,iMessage,wParam,lParam));
    }
        return 0L;
}
```

2.6.2 Потокова багатозадачність

Потік – це частина процесу, який виконується паралельно з виконанням інших частин процесу. В рамках одного процесу Windows спочатку створює тільки один потік, який надалі будемо називати головним.

Якщо деякі частини програми можуть виконуватись одночасно (наприклад, введення даних, обчислення, друк на принтері), тоді програміст може передбачити в програмі створення додаткових потоків.

Всі потоки одного процесу знаходяться в спільному просторі пам'яті цього процесу і спільно використовують виділені процесу ресурси. Важливо зрозуміти, як потоки використовують найголовніший спільний ресурс – центральний процесор. Кожному потокові виділяється квант часу (в Windows-95 квант часу рівний 20 мс) протягом якого йому повністю надається центральний процесор. Потік або виконується до закінчення виділеного йому кванта часу, або його виконання може бути перерване якою-небудь подією, наприклад необхідністю доступу до зайнятого ресурсу чи необхідністю одержання результатів роботи інших потоків.

Швидке перемикання між потоками створює враження їх паралельної роботи. Розв'язання проблем, що постійно виникають при перемиканні потоків, накладаються на операційну систему.

Обов'язок програміста полягає лише в написанні програмного коду для формування нових потоків в доповнення до головного потоку процесу.

Як правило, окремий потік створюється для виконання одної функції програми (в даному випадку під функцією розуміється власна функція програміста, а не системна).

В принципі, на основі всіх власних функцій програми можна створити відповідні потоки. Зрозуміло, що створювати окремий потік є сенс лише для функцій з тривалим часом виконання – завантаженням бітового зображення чи друком на принтері.

Для створення окремого потоку призначена API-функція *CreateThread*(), формат якої такий:

```
HANDLE CreateThread(LPSECURITY_ATTRIBUTES lpAttr,
                    DWORD dwStack,
                    LPTHREAD_START_ROUTINE lpFunc,
                    LPVOID lpParam, DWORD dwFlags,
                    LPDWORD lpdwID );.
```

В цій функції параметр *lpAttr* – покажчик на структуру атрибутів доступу, що надаються створюваному процесу. Для Windows 95 цей параметр має бути рівний *NULL*. Параметр *dwStack* – це розмір стека потоку. Якщо *dwStack= NULL*, тоді розмір стека дорівнює розміру стека батьківського процесу.

Параметр *lpFunc* є адресою потокової функції. Всі потокові функції повинні мати такий прототип:

```
DWORD threadfunc(LPVOID lpPparam);.
```

В потоковій функції параметр *lpParam* визначається при створенні потоку в API-функції *CreateThread*().

Параметр *dwFlags* означає стан потоку. Якщо *dwFlags*=0, тоді новий потік починає виконуватись негайно. Якщо *dwFlags=CREATE_SYSPEND*.

тоді потік буде знаходитись в стані очікування до появи сигналу дозволу від API-функції *ResumeThread*().

Потік закінчується при поверненні із потокової функції. Батьківський процес може примусово припинити його за допомогою виклику API-функції *TerminateThread*():

BOOL TerminateThread(HANDLE hThread, DWORD Status); .

В цій функції параметр *hThread* означає дескриптор потоку, що припиняється, а параметр *Status* – код завершення.

API-функції *CreateThread*() та *TerminateThread*() повертають ненульове значення при успішному завершенні, і нульове – в іншому випадку.

2.6.3 Синхронізація потоків

Синхронізація необхідна для забезпечення доступу кількох процесів або потоків до одного ресурсу, який може використовуватись лише одним потоком. Наприклад, якщо один потік записує інформацію у файл, то інші потоки не можуть в даний момент часу використовувати цей файл. Механізм запобігання таких ситуацій називається серіалізацією.

Windows-95 підтримує чотири типи об'єктів синхронізації:

- класичні семафори;
- двійкові семафори;
- критичні секції;
- події.

Класичний семафор (далі – просто семафор) регулює доступ кількох процесів або потоків до спільного ресурсу. Операційна система створює семафор при створенні головного вікна програми. Програміст може створити власні семафори за допомогою API-функції *CreateSemaphore*():

HANDLE *CreateSemaphore*(LPSECURITY_ATTRIBUTES lpAttr,

LONG InitialCount, LONG MaxCount,LPSTR lpszName); .

де *InitialCount* – початкове значення лічильника семафора;

MaxCount – максимальне значення лічильника семафора, тобто максимальна кількість потоків, що можуть одночасно працювати з ресурсом ;

lpszName - покажчик на рядок з іменем семафора;

lpAttr – атрибут доступу, який для Windows-95 має бути рівний NULL.

Отримавши доступ до ресурсу, потік збільшує значення лічильника семафора на одиницю. Якщо після цього лічильник досягне значення *MaxCount*, тоді всі інші потоки, які бажають отримати доступ до даного ресурсу, переходять в режим очікування. Для забезпечення режиму очікування семафора іншими потоками використовується API-функція *WaitForSingleObject*():

DWORD WaitForSingleObject(HANDLE hObject, DWORD dwTime); .

де *dwTime* – час очікування (в мсек.) доступу до об'єкта *hObject*.

Якщо потік протягом заданого періоду часу очікування не отримує доступу до об'єкта (в даному випадку – семафора) тоді функція повертає значення WAIT_TIMEOUT. При успішному завершенні ця функція повертає значення WAIT_OBJECT_0.

Після закінчення роботи з ресурсом потік повинен звільнити семафор. Для такої операції передбачена API-функція *ReleaseSemaphore*( ):

BOOL ReleaseSemaphore(HANDLE hSemaphore, LONG Count,
LPLONG lpPrevCount); ,

де *hSemaphore* – дескриптор семафора;

Count – величина, на яку має змінитися значення лічильника, як правило, на 1;

lpPrevCount – покажчик на змінну, куди записується попереднє значення лічильника семафора, якщо така інформація не потрібна, тоді цей параметр рівний NULL.

Якщо максимальне значення лічильника семафора не перевищує 1, тоді такий семафор працює як двійковий, надаючи доступ до ресурсу тільки одному потокові. Однак в Windows-95 передбачений також і спеціальний двійковий семафор – mutex..

Серед інших об'єктів синхронізації Windows-95 також часто використовуються події, які дозволяють повідомити процесам або потокам про виникнення визначеної ситуації. Для створення об'єкта події використовується API-функція *CreateEvent*():

*HANDLE CreateEvent(*LPSECURITY_ATTRIBUTES lpAttr,
BOOL Manual, BOOL Initial, .LPSTR lpszName); ,

де Manual – визначення поведінки об'єкта після настання події, якщо це значення дорівнює TRUE, тоді подія буде відмінена лише за допомогою виклику функції *ResetEvent*( );

Initial – початкове значення події.

Інші параметри функції API-*CreateEvent*() мають той же смисл, що і для API-функції *CreateSemaphore*().

Після створення об'єкта події процес або потік, який чекає на цю подію, викликає API-функцію *WaitForSingleObject*(), задаючи першим параметром дескриптор цієї події. В результаті виконання процесу або потоку об'єкт події переривається до настання цієї події. Для повідомлення про настання події з дескриптором *hEventObject*, яку чекають, використовується API-функція *SetEvent*():

BOOL *SetEvent(HANDLE hEventObject);* .

Четвертий тип об'єктів синхронізації Windows-95 – критична секція – дозволяє визначити деяку область коду програми, що спільно використовується кількома потоками, причому не більше одного потоку може мати одночасний доступ до цієї області.

Тепер розглянемо програму, в якій створено два додаткових потоки. Перший потік створюється під час виконання функції *MyThreadFunc1*, а другий потік – функції *MyThreadFunc2*. Обидва потоки виконуються по черзі. Кожен потік видає на екран свій ідентифікатор і проміжний результат обчислень. Керує доступом потоків до процесора семафор, що створюється при створенні головного вікна програми.

```
#include <windows.h>
#include <stdio.h>
#include <string.h>
#include "Resource.h"
DWORD  MyThreadFunc1(LPVOID);
DWORD  MyThreadFunc2(LPVOID);
 LRESULT CALLBACK WindowFunc(HWND,UINT,UINT,LPARAM);

 char str[80];
 HDC hdc;
DWORD Tid1,Tid2;
HANDLE hSemaphore;
int  X=0, Y=0;
int j=0;
 BOOL MyInit(HINSTANCE hInstance)
 {
  WNDCLASS WndClass;
  BOOL bSuccess;
  WndClass.style=NULL;
  WndClass.lpfnWndProc=WindowFunc;
  WndClass.hInstance=hInstance;
  WndClass.cbClsExtra=NULL;
  WndClass.cbWndExtra=NULL;
  WndClass.hIcon=LoadIcon(NULL,IDI_APPLICATION);
  WndClass.hCursor=LoadCursor(NULL,IDC_ARROW);
  WndClass.hbrBackground=(HBRUSH)GetStockObject(WHITE_BRUSH);
  WndClass.lpszMenuName="MyMenu";
  WndClass.lpszClassName=(LPSTR)"MyClass";
    bSuccess=RegisterClass(&WndClass);
            return bSuccess;
}
int WINAPI WinMain(HINSTANCE hInstance,
       HINSTANCE hPrevInstance, LPSTR lpszCmdLine,int nCmdShow)
{
      HWND hWnd;
      MSG msg;
      if(!hPrevInstance)
```

```
        if(!MyInit(hInstance))
          return NULL;
hWnd=CreateWindow("MyClass",
"MyProgram",
WS_OVERLAPPEDWINDOW,
CW_USEDEFAULT,
CW_USEDEFAULT,
CW_USEDEFAULT,
CW_USEDEFAULT,
NULL,
NULL,
hInstance,
NULL);
if( ! hWnd)
     return NULL;
ShowWindow(hWnd,nCmdShow);
UpdateWindow(hWnd);
  while(GetMessage(&msg, NULL, NULL, NULL))
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
                  return msg.wParam;
}

LRESULT CALLBACK WindowFunc(HWND hWnd,UINT iMessage,
                        WPARAM wParam,LPARAM lParam)
{
STARTUPINFO StartupInfo;
PROCESS_INFORMATION  ProcessInfo;
     switch(iMessage)
 {
  case WM_COMMAND:
   switch(wParam)
   {
     case IDM_THREAD:
           {
                        hdc=GetDC(hWnd);
    CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)MyThreadFunc1,
                        (LPVOID)hWnd,0,&Tid1);
    CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)MyThreadFunc2,
                           (LPVOID)hWnd,0,&Tid2);
              ReleaseDC(hWnd,hdc);
   }    break;
```

```
    case IDM_PROCESS:
    memset(&StartupInfo,0,sizeof(StartupInfo));
        StartupInfo.cb=sizeof(STARTUPINFO);
    CreateProcess(NULL,"c:\\test.exe",
          NULL,NULL,FALSE,0,NULL,NULL,
          &StartupInfo,&ProcessInfo);
           }     break;
       case WM_DESTROY:
        PostQuitMessage(0); break;
           default:
      return(DefWindowProc(hWnd,iMessage,wParam,lParam));
    }
          return 0L;
}
DWORD  MyThreadFunc1(LPVOID Param)
{
      int i;     DWORD curTid=Tid1;
      if(WaitForSingleObject(hSemaphore,1000)==WAIT_TIMEOUT)
      {
MessageBox((HWND)Param,"Thread1 - error", "Error semapfore",MB_OK);
            return 0;
      }
       for(i=0; i<5; i++)
       {      Sleep(2000);
              sprintf(str,"Thread1=%d, signal %d",curTid,i);
              TextOut(hdc,X,Y+j,str,strlen(str)); j=j+15;          }
        ReleaseSemaphore(hSemaphore,1,NULL);
      return 0;
}
DWORD  MyThreadFunc2(LPVOID Param)
{
      int i;     DWORD curTid=Tid2;
      if(WaitForSingleObject(hSemaphore,1000)==WAIT_TIMEOUT)
      {
MessageBox((HWND)Param,"Thread2 - error", "Error semapfore",MB_OK);
            return 0;
      }
       for(i=0; i<5; i++)
       {      Sleep(2000);
              sprintf(str,"Thread2=%d, signal %d",curTid,i);
              TextOut(hdc,X,Y+j,str,strlen(str)); j=j+15;          }
        ReleaseSemaphore(hSemaphore,1,NULL);
      return 0;
}
```

3 Основи об'єктно-орієнтованого програмування мовою С++

3.1. Інкапсуляція даних

Інкапсуляція - це механізм, який об'єднує дані і код, що працює з цими даними і захищає перше та друге від зовнішнього втручання або неправильного використання. В об'єктно-орієнтованому програмуванні дані і функції (тобто код) об'єднуються разом і створюють абстрактні типи даних, тобто типи даних, які визначаються користувачем. В термінології мови Сі++ абстрактні типи даних називають класами.

При конкретизації даних та функцій класу створюються об'єкти. Клас служить як шаблон, на основі якого може бути створена велика кількість різних об'єктів.

Дані, які належать об'єкту даного класу, обумовлюють стан даного об'єкта, а набір функцій, тобто код, обумовлює поведінку об'єктів даного класу.

3.1.1 Описання класу

Розглянемо більш детально поняття класу.

Клас - абстрактний тип даних, що визначається користувачем. Він може містити опис стандартних типів даних і операцій над ними, тобто одночасно можуть знаходитись і змінні різних типів, і функції з цими змінними.

Формально найпростіше описання класу виглядає так:

ключове_слово < ім'я_класу>

{

список_членів_класу

}.

В полі "*ключове_слово*" записується одне з таких слів: *class, struct, union.*

Поле "*ім'я_класу*" - унікальний ідентифікатор класу. Якщо в поле "*ключове_слово*" записані ключові слова *struct* чи *union*, то ім'я класу може бути відсутнім. Поле "*список_членів_класу*" містить описання типів і імен як даних, так і функцій, що називаються також функціями-членами. Таке співіснування даних і функцій, що працюють з цими даними, називають інкапсуляцією.

Дані, що належать об'єктам деякого класу, обумовлюють стан даного об'єкта, а набір функцій-членів обумовлює поведінку об'єктів класу.

Якщо розглядати виклик функції-члена як запит до об'єкта для виконання певних дій, то можна сказати, що набір функцій-членів визначає множину запитів, на які об'єкти даного класу будуть реагувати відповідним чином.

Найважливішою особливістю абстрактних типів даних є можливість задання різних режимів доступу (областей видимості) класу. В C++ режими доступу задаються за допомогою таких атрибутів:

- *public* (загальнодоступний);
- *protected* (захищений);
- *private* (власний).

Атрибут "*public*" робить дані і функції доступними (видимими) з будь-яких точок програми. Атрибут "*protected*" робить дані і функції доступними тільки для даного класу чи для похідних класів (класів-нащадків).

За замовчуванням, тобто при відсутності в назві класу вказаних атрибутів, режим доступу визначається видом класу. Якщо клас визначається ключовим словом *class*, то його члени за замовчуванням будуть вважатись закритими (атрибут "*private*"). Наприклад, об'єкти класу "*книга*" можна описати таким чином:

```
class book
{
    char author[20];
    char title[25];
    int year;
public:
    float price;
    void Set();
    void Print();
};
```

В даному прикладі змінні *author[20]*, *title[25]*, *year* є закритими, тобто власними даними класу *book*, а змінна *price* і функції-члени *Set()* та *Print()* є загальнодоступними.

Слід відзначити особливості описання таких видів класів як структури (визначаються ключовим словом "*struct*") і об'єднання (визначаються ключовим словом "*union*"). Структура - це такий клас, в якому відсутні функції, а всі дані мають за замовчуванням атрибут "*public*".

Наприклад:

```
struct book
{
    char author[20];
    char title[25];
    int year;
    float price;
};
```

При необхідності, в структурі можна використовувати атрибут "*private*".

Об'єднання за синтаксисом дуже схожі на структури, однак мають дуже суттєві обмеження: вони не можуть успадковувати інші класи та не можуть самі використовуватись як базовий клас.

3.1.2 Описання функцій у класі

Функції, що належать класу, можуть бути описані двома способами: внутрішнім і зовнішнім.

При внутрішньому описанні тіло функції розташовується всередині декларації класу. Цей спосіб варто використовувати для дуже коротких функцій. При зовнішньому описанні тіло функції розміщується за межами класу, а для вказування її зв'язку з класом використовується оператор (::) .

Вибір між внутрішнім та зовнішнім описанням функції визначає спосіб її компіляції: згенеровані машинні команди або безпосередньо підставляються в об'єктний модуль подібно до макроса (при внутрішньому описанні), або оформлюються як функція, що викликається.

Як приклад розглянемо описання класу *ADDRESS*, в якому функція *Get()* визначається всередині класу, а функція *Print()* - за межами класу.

```
class ADDRESS
{
    char *street; /* вулиця */
    int   house;  /* номер будинку */
public:
    char *town; /* місто */
    void Print();
    void Get()
{
cout << "Введіть адресу " << "\n";
cin >> town;
cin >> street;
cin >> house;
}
};
void ADDRESS::Print()
{
cout << "\n";
cout << "town:"; puts(town);
cout << "street:"; puts(street);
cout << "house:" << house;
}.
```

Описання класу ще не створює об'єктів даного класу. Об'єкти створюються тільки при описанні відповідних їм змінних, наприклад:

```
ADDRESS adr1,adr2,adr3; .
```

При роботі з багатомодульними програмами описання класу повинно бути присутнім у кожному модулі, де використовуються об'єкти даного класу або визначаються його функції-члени. Тому має сенс розташовувати описання класу у заголовочному файлі, який приєднується за допомогою директиви *#include* в ті модулі, де він потрібний.

Доступ до відкритих членів об'єкта деякого класу здійснюється за допомогою операторів прямого (символ ".") чи опосередкованого (символ "->") вибору.

Нехай описання розглянутого вище класу *ADDRESS* розташовано в заголовочному файлі "*address.h*". Тоді основна програма, в якій створюється масив з десяти об'єктів класу *ADDRESS* і виводяться на друк об'єкти, де використовується назва міста "Вінниця", має вигляд:

```
#include <iostream.h>
#include <stdio.h>
#include <string.h>
#include "address.h"
void main()
{
      ADDRESS adr[10]; int i;
      for (i=0;i<10;i++)
       adr[i].Get();
      for(i=0;i<10;i++)
      {
       if (strcmp(adr[i].town,"Вінниця"))
         puts(" ");
       else adr[i].Print();
      }
}
```

3.1.3 Конструктори і деструктори

Серед функцій-членів класу є такі, які визначають особливості створення, ініціалізації, копіювання та знищення об'єктів даного класу. Конструктори і деструктори - приклад важливих спеціальних функцій-членів.

Функція, основна мета якої - ініціалізація змінних об'єкта даного класу або розподіл пам'яті для їх зберігання, називається конструктором. Конструктор має таке ж ім'я, як і клас, в якому він визначений. З об'єктом завжди пов'язано або явне, або опосередковане виконання конструктора. Якщо відсутній явно описаний конструктор, тоді автоматично генерується так званий конструктор за замовчуванням. Конструктор не може бути викликаний явно з-за меж програми. Він викликається явно компілятором при створенні об'єкта і неявно при виконанні оператора *new*, що застосовується до об'єкта, а також при копіюванні об'єкта даного класу.

Функція-деструктор знищує об'єкт даного класу. Ім'я деструктора утворюється шляхом додавання символу ~ (тільда) на початку назви класу. Деструктор викликається автоматично при знищенні об'єкта, наприклад, при завершенні програми. Явне знищення об'єкта виконує оператор *delete*.

Наведемо приклад програми, в якій ініціалізація об'єктів класу *ADDRESS* виконується за допомогою конструктора.

```
#include <iostream.h>
#include <stdio.h>
#include <conio.h>
class ADDRESS
{
        char *town;
        char *street;
        int house;
    public:
        ADRESS(char *, char *, int);
        ~ADDRESS();
        void Print();
};
    ADDRESS::ADDRESS( char *t, char *s, int h)
    {
    town=t;  street=s;  house=h;
    };
    ADDRESS::~ADDRESS()
    {
    cout << "\n" << "Delete object";
    };
    void ADDRESS::Print()
    {
   cout << "\n";
    cout << "town: " <<town << endl;
    cout << "street:"<<street<<endl;
    cout << "house:" << house;
    };
void main()
{
    ADDRESS  adr1("Kiev","Naumova",25);
    ADDRESS  adr2("Vinnitsa","Pavlova",48);
    clrscr();
    adr1.Print();
    adr1.Print();
   getch();
}
```

3.2 Успадкування

Успадкування - це процес, за допомогою якого один об'єкт може набувати властивостей іншого. Точніше, об'єкт може успадковувати основні властивості іншого об'єкта і додавати до нього риси, характерні лише для нього.

За допомогою успадкування можна створювати дуже складні класи, рухаючись від простих за структурою класів.

Створюючи нові класи, можна підтримувати ієрархію класів, що дозволяє керувати великими потоками даних.

3.2.1 Базові і похідні класи

Клас може успадковувати риси функціонування іншого класу як похідний від нього.

Існуючий клас, який служить основою для створення нового класу, називається базовим. Відповідно новий клас називається похідним.

Будь-який клас може бути використаний як базовий, і будь-який похідний клас відображає опис базового класу.

Похідний клас може модифікувати права доступу до даних класу, додавати нові члени чи "перевантажувати" існуючі функції.

Опис похідного класу має такий синтаксис:

ключове_слово ім'я_похідного_класу: базовий_список
{
список_членів_класу,
};

де поле "*ключове_слово*" може містити *class*, *struct* чи *union*; поле "*ім'я_похідного_класу*" - ідентифікатор похідного класу; поле "*базовий_список*" містить перелік модифікаторів доступу, що розділені між собою комою (*public* чи *private*) , і імен базових класів.

Похідний клас успадковує всі члени перерахованих базових класів, але може використовувати тільки члени базових класів з атрибутом *public* чи *protected*.

Модифікатори доступу не впливають на атрибути доступу базового класу, але можуть їх змінювати при звертанні до членів базового класу з боку похідного класу (тільки в напрямку більшого обмеження доступу).

Розглянемо простий приклад успадкування властивостей.

Нехай існує клас *POINT*, в якому визначені координати точки на екрані дисплея і функція-конструктор для присвоєння значень цим координатам:

```
class POINT
{
   int X,Y;
public:
```

```
    POINT(int, int);
};
POINT::POINT(int NewX ,int NewY)
{
X=NewX; Y=NewY;
}.
```

Визначимо новий клас *PIXEL*, що характеризує не тільки положення точки на площині, але й її колір. Як базовий клас візьмемо клас *POINT*. Тоді похідний клас *PIXEL* буде описуватись таким чином:

```
class PIXEL:public POINT
{
        int Color;
public:
    PIXEL(int, int, int);
};
    PIXEL::PIXEL(int NewX, int NewY, int NewColor):
                Point(NewX, NewY)
    {
    Color=NewColor;
    putpixel(Newx,NewY,Color);
    }.
```

Модифікатор доступу public в полі "*базовий_список*" класу *PIXEL* не змінює атрибутів членів класу *POINT*, тому конструктор класу *PIXEL* визначається через доступний йому конструктор класу *POINT*.

В залежності від значення статусів доступу в базовому і похідному класах можна отримати багато варіантів прав доступу. Можна, наприклад, зробити доступними для класу *PIXEL* всі члени класу *POINT*. Тоді можна обійтись без звертання до конструктора класу *POINT*, що дозволить прискорити роботу програми.

```
class POINT
    {
      protected:
     int X, Y;
public:
    POINT(int, int);
};
 POINT::POINT(int NewX, int NewY)
 {
 X=NewX; Y=NewY;
 };
```

```
class PIXEL:public POINT
    {
       int Color;
    public:
        PIXEL(int, int, int);
    };
    PIXEL::PIXEL(int NewX, int NewY, int NewColor)
    {
       X=NewX;
       Y=NewY;
       Color=NewColor;
       putpixel(NewX,NewY,Color);
    }.
```

3.2.2 Доступ до захищених членів базового класу

Можливі декілька варіантів успадкування членів базового класу.

Якщо специфікатором доступу базового класу є *public*, то всі відкриті члени базового класу стають відкритими і в похідному. Якщо специфікатором доступу є *protected*, тоді всі захищені члени базового класу доступні для всіх похідних від нього класів. За межами базового або похідних класів захищені члени недоступні. Нарешті, якщо базовий клас успадковується як *private*, тоді всі члени базового класу стають недоступними і для похідних класів.

Доступ до закритих членів базового класу можна отримати, якщо в базовому класі існують функції з специфікатором доступу *public* або *protected*. У такому випадку, з одного боку, такі функції мають доступ до всіх членів свого класу, а з іншого – самі ж функції є доступними для похідних класів. В результаті такого непрямого шляху можна оперувати закритими членами базового класу.

В наступній програмі реалізований непрямий доступ. В класі *BOOK* є відкрита функція *GetYear*(), що повертає значення закритого члену *year* базового класу. До такої функції можна звернутися не тільки із похідних класів, але й із функції *main*().

```
#include <iostream.h>
#include <conio.h>
class BOOK
{
      char * author;
      char *title;
      int year;
public:
      BOOK(char *, char*, int);
```

```
        void Print();
        int GetYear();
};
BOOK::BOOK(char *a, char *t, int y)
{
        author=a;  title=t;  year=y;
}
int BOOK::GetYear()
{
        return(year);
}
void BOOK::Print()
{
        cout << "author: " << author << endl;
        cout <<  " title: " << title << endl;
        cout <<  " year: " << year  <<  endl;
}

void main()
{
        int i,  mas[10];
        char str1[10][20],   str2[10][20];
        BOOK  *library[10];
                clrscr();

        for(i=0; i<10; i++)
        {
        cout << "\n № " << i+1
        cout << " Author " << i+1 << ":";
        cin >> str1[i];
        cout << "Title " << i+1 << ":";
        cin >> str2[i];
        cout << "Year " <<  i+1 << ":";
        cin >> mas[i];
        library[i]=new BOOK(str1[i], str2[i], mas[i]);
        }
        cout << endl;
        for(i=0; i<10; i++)
        {
        if(library[i]->GetYear() >= 1995)
        library[i]->Print();
        }
}
```

3.2.3 Ієрархія класів

Похідний клас може стати базовим по відношенню до створюваного нового класу, останній може стати базовим по відношенню до наступного і т.д. В результаті створюється ієрархія класів, яку зручно зображати ієрархічним деревом, в вершині якого розташовується базовий клас, а знизу - похідні класи.

Як приклад розглянемо програму для рисування ліній і кіл заданих кольорів на екрані дисплея. Введемо класи LINE та CIRCLE для отримання відповідно об'єктів - горизонтальної лінії та кола на основі раніш розглянутих класів. В цьому випадку ієрарихія класів буде мати вигляд (рис. 3.1).

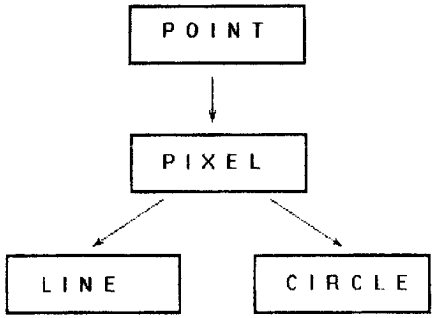


Рисунок 3.1 - Приклад ієрархії класів

Повний текст програми, що реалізує таку ієрархію класів, буде мати такий вигляд.

```
#include <iostream.h>
#include <conio.h>
#include <graphics.h>
class POINT
{
        int x,y;
    public:
        POINT(int, int);
};
    POINT::POINT(int NewX, int NewY);
    {
    X=NewX; Y=NewY;
    };
```

```
class PIXEL: public POINT
{
        int color;
    public:
      PIXEL(int, int, int);
};
    PIXEL::PIXEL(int NewX, int NewY, int NewColor):
                                POINT(NewX, NewY)
    {
    Color=NewColor;
    putpixel(NewX, NewY,Color);
    };
class CIRCLE : public PIXEL
{
        int Radius;
    public:
        CIRCLE(int, int, int, int);
};
    CIRCLE::CIRCLE(int NewX, int  NewY,int
    NewColor, int NewRadius) : PIXEL(NewX, NewY, NewColor)
    {
    Radius=NewRadius;
    setcolor(NewColor);
    circle(NewX, NewY,Radius);
    };
class LINE : public PIXEL
{
        int Vector;
    public:
        LINE(int, int, int, int);
};
    LINE::LINE(int NewX, int NewY, int NewVector, int    NewColor ):
                                        PIXEL(NewX,NewY,NewColor)

    {
    Vector=NewVector;
    for(int i=NewX;i<Vector;i++)
     PIXEL(i,NewY,NewColor);
    };
void main()
{
    int driver=DETECT,mode;
    initgraph(&driver,&mode,"d:\\borlandc\\bgi");
    PIXEL pixelA(20,100,12);
```

```
    CIRCLE circleM(200,300,12,50);
    LINE lineN(120,150,200,3);
    getch();
    closegraph();
}.
```

В результаті виконання цієї програми на екрані дисплея з'явиться червона точка з координатами: X=20 та Y=100, червоне коло з координатами центра X=120 та Y=150 і з радіусом 50, а також горизонтальна лінія блакитного кольору довжиною 200, що починається у точці з координатами X=120 та Y=150.

3.2.4 Множинне успадкування

Похідний клас може мати будь-яку кількість базових класів. Використання двох та більше класів називається множинним успадкуванням.

Розглянемо клас *MCIRCLE*, який дозволяє виводити на екран повідомлення всередині кола. Раніше вже був представлений клас CIRCLE для рисування кола. Введемо спеціальний клас MESSAGE, що дозволяє виводити на екран повідомлення в графічному режимі роботи дисплея.

```
class MESSAGE : public PIXEL
{
    int Font,Size; char *Msg;
    public:
    MESSAGE(int,int,int,int,int,char *);
    };
    MESSAGE::MESSAGE(int NewX,int NewY, int NewColor,
     int NewFont,int NewSize, char *NewMsg):
     PIXEL(NewX,NewY,NewColor)
    {
    Font=NewFont; Size=NewSize; Msg=NewMsg;
    setcolor(NewColor);
    settextstyle(Font,0,Size);
    settextjustify(NewX,NewY);
    outtextxy(NewX,NewY,Msg);
    }.
```

Тепер можна ввести клас *MCIRCLE*, для якого базовим класом будуть класи *CIRCLE* і *MESSAGE*:

```
class MCIRCLE : CIRCLE,MESSAGE
{
    public:
    MCIRCLE(int,int,int,int,int,int,char *);
};
    MCIRCLE::MCIRCLE(int MyX,int MyY,int MyColor,
```

```
    int MyRadius, int MyFont,int MySize, char *MyMsg):
      CIRCLE(MyX,MyY,MyColor,MyRadius),
    MESSAGE(MyX,MyY,MyColor,MyFont,MySize,MyMsg)
    {
    };
```

Клас *MCIRCLE* ніяк не розширює існуючі базові класи, а лише об'єднує їх можливості. Як видно з опису класу *MCIRCLE*, імена членів класу можуть бути довільними, аби лише вони збігались у похідному та базовому класах, також не можна порушувати порядок переліку членів класу *MCIRCLE* з раніш введеними класами *CIRCLE* і *MESSAGE*.

На рис. 3.2 показана ієрархія класів для розглянутого варіанта множинного успадкування.

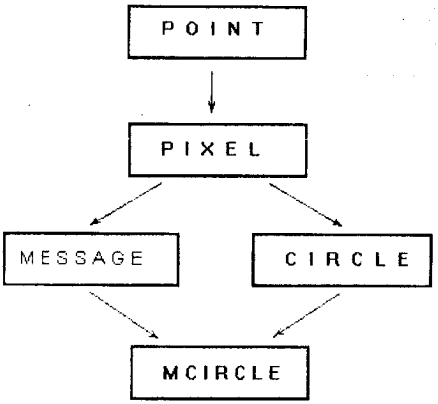


Рисунок 3.2 - Приклад множинного успадкування

Якщо на екрані дисплея необхідно нарисувати коло червоного кольору радіусом 75 із центром в точці X=150, Y=150, а всередині кола вивести слово "*TEST*", тоді відповідний об'єкт *Text* класу *MCIRCLE* створюється за допомогою такої програми:

```
#include <graphics.h>
#include <conio.h>
void main()
{
    char str[]="test";   int driver=DETECT, mode;
    initgraph(&driver ,&mode," ");
    MCIRCLE  text(150,150,12,75,1,1,str);
    getch();
    closegraph();
}
```

3.2.5 Дружні класи

Однією з найважливіших властивостей класу є можливість "приховування" даних. Однак в деяких випадках бажано, щоб функція, що не є членом класу, мала доступ до закритих членів даного класу. Цього можна досягти, якщо оголосити таку функцію "*другом*" класу за допомогою ключового слова *friend*. Аналогічно можна оголосити деякі класи дружніми за допомогою ключового слова *friend*.

Розглянемо останній випадок більш докладно.

Нехай розглянутий вище клас *PIXEL* є базовим по відношенню до деякого класу *LINE* і до класу *BAR*.

Об'єкти класу *LINE* рисують на екрані горизонтальну лінію заданого розміру і кольору, а об'єкти класу *BAR* - заповнений прямокутник заданого розміру. Якщо необхідно, щоб об'єкти класу *BAR* також могли рисувати горизонтальні лінії, тоді є сенс скористатись властивостями класу *LINE*. Як правило "контакти" між сусідніми похідними класами, що мають спільний базовий клас, заборонені. Щоб обійти вказану заборону, слід класи *LINE* та *BAR* оголосити дружніми, як показано нижче.

Тоді в функції *main*() можна створити об'єкт, що дозволяє рисувати лінію і прямокутник:

```
BAR barN(120,150,200,300,3);
```

Важливо відзначити, що у функції *main*() функція

```
LINE LineM(100,100,312,3);
```

буде недоступною.

Повний текст програми, що використовує принцип дружніх класів, показаний нижче.

```
#include <iostream.h>
#include <conio.h>
#include <graphics.h>
class POINT
{
        int x ,y;
    public:
        POINT(int ,int);
};
    POINT::POINT(int NewX, int NewY)
    {
    X=NewX;
    Y=NewY;
    };
class PIXEL : public POINT
{
        int color;
    public:
```

```
    PIXEL(int, int, int);
};
PIXEL::PIXEL(int NewX, int NewY, int NewColor):
      POINT(NewX, NewY)
{
Color=NewColor;
putpixel(NewX, NewY, Color);
};
class LINE : public PIXEL
{
   int Vector;
private:
   LINE(int, int, int, int);
friend class BAR;
};
LINE::LINE(int NewX, int NewY, int NewVector, int  NewColor) :
    PIXEL(NewX, NewY, NewColor)
{
Vector=NewVector;
setcolor(NewColor);
for(int i=NewX; i<Vector;i++);
PIXEL(i, NewY, NewColor);
};
class BAR : public PIXEL
{
   int X2,Y2;
public:
   BAR(int, int, int, int, int)
friend class LINE;
};

BAR::BAR(int X1, int Y1, int NewX2, int NewY2, int NewColor):
                PIXEL(X1, Y1, NewColor)
{
 X2=NewX2; Y2=NewY2;
for(int i=X1;i<NewX2;i++)
for(int j=Y1;j<NewY2;j++)
PIXEL(i,j,NewColor);

LINE(X1,Y1-50, 125, NewColor);
};

void main()
{
```

```
    int driver=DETECT,mode;
    initgraph(&driver,&mode,"d:\\borlandc\\bgi");
    // LINE lineM(100,100,312,2);  ця функція недоступна
    BAR barN(120,150,200,300,3);
    getch();
    closegraph();
}.
```

Необхідно відзначити, що дружні класи дозволяють створювати горизонтальні зв'язки в ієрархії класів, а похідні класи - вертикальні зв'язки.

3.3 Поліморфізм

Поліморфізм - це властивість, яка дозволяє одне і те ж ім'я використовувати для двох чи більше різних задач.

Основною ідеєю поліморфізму є "єдиний інтерфейс", або множина методів. Це означає, що можна створювати загальний інтерфейс для групи близьких за змістом дій. При цьому виконання конкретної дії залежить від даних. Перевагою поліморфізму є те, що він допомагає знижувати складність програм, дозволяючи використання одного інтерфейсу (тобто одного опису) для задання єдиного класу дій. Вибір же конкретної дії, в залежності від ситуації, покладається на компілятор; програмісту не потрібно робити цей вибір самому.

На мові С++ поліморфізм реалізується за допомогою механізму віртуальних функцій. Окремими випадками поліморфізму є також перевантаження функцій та перевантаження операторів.

3.3.1 Віртуальні функції

Як приклад розглянемо текст програми, яка описує керування функціональними пристроями (клавіатурою, дисплеєм, таймером та ін.) персонального комп'ютера.

Створимо базовий клас *UNIT* для загального описування функціональних пристроїв комп'ютера. Обмежимось тільки загальною характеристикою вказаних пристроїв, а саме тим, що всі вони керуються системною програмою-драйвером:

```
class UNIT
{
    public:
    void driver()
    { }
};
```

В класі *UNIT* визначена єдина існуюча функція *driver*(), яка не виконує в даному випадку ніяких дій. Її присутність лише свідчить про присутність драйвера для кожного пристрою.

Конкретний вигляд драйвера необхідно задати при описанні класу конкретного пристрою. Створимо класи для таких пристроїв, як клавіатура, дисплей і таймер, використовуючи механізм успадкування:

```
class KEYBOARD : public UNIT
{
      void driver()
      {
      puts("Драйвер клавіатури - KEYBOARD.DRV");
      }
};
class DISPLAY:public UNIT
{
      void driver()
      {
      puts("Драйвер дисплея - SUPERVGA.DRV");
      }
};
class TIMER : public UNIT
{
      void driver()
      {
      puts("Драйвер таймера - SYSTEM.DRV");
      }
};.
```

Як видно з прикладу, оголошена в базовому класі функція *driver*() перевизначається в кожному похідному класі. Для того, щоб компілятор міг вірно викликати фукнцію з однаковим іменем і визначену в декількох класах, необхідно оголосити її віртуальною в базовому класі. З цією метою перед типом функції ставиться ключове слово *virtual*:

```
class UNIT
{
      public:
      virtual void driver()
      { }
};.
```

В результаті, віртуальна функція всередині базового класу задає "інтерфейс" цієї функції, а її перевизначення в похідних класах задає "множину методів", тобто множину конкретних реалізацій цієї функції.

При перевизначенні віртуальної функції в похідному класі ключове слово "*virtual*" вказувати не потрібно.

3.3.2 Покажчики на похідні класи

Віртуальна функція може викликатись так само, як і будь-яка інша функція-член. Однак найбільш цікавий виклик функції через покажчик, завдяки чому підтримується динамічний поліморфізм.

В мові C++ покажчик, оголошений як покажчик на базовий клас, також може використовуватись і як покажчик на будь-який клас, похідний від даного базового. Пояснюється це тим, що базовий покажчик "знає" тільки про базовий клас і нічого "не знає" про члени, які додались до базового класу.

Покажчик базового класу можна використовувати для вказування на об'єкт похідного класу, але зворотній порядок - недійсний.

Варто також пам'ятати, що арифметика покажчиків пов'язана із типом даних, який заданий при оголошенні покажчика. Це означає, що коли базовий клас вказує на об'єкт похідного класу, а потім інкрементується, то він вже не буде вказувати на наступний об'єкт похідного класу. Цей покажчик буде вказувати на наступний об'єкт базового класу.

Як приклад застосуємо покажчики в функції *main*() для раніш розглянутих класів.

```
void main()
{
      UNIT *ptr;
      KEYBOARD keyboard1;
      DISPLAY display1;
      TIMER timer1;
      ptr=&keyboard1;
      ptr->driver();
      ptr=&display1;
      ptr->driver();
      ptr=&timer1;
     ptr->driver();
}.
```

Після виконання даної програми на екран дисплея буде виведено:

```
Драйвер клавіатури - KEYBOARD.DRV
Драйвер дисплея - SUPERVGA.DRV
Драйвер таймера - SYSTEM.DRV .
```

3.3.3 Абстрактні класи

Віртуальна функція, оголошена в базовому класі, часто не виконує ніяких корисних дій. Така функція називається чисто віртуальною функцією, в базовому класі вона повинна обов'язково перевизначатись. В базовому класі задається тільки прототип чисто віртуальної функції такого виду:

virtual_тип_ім'я_функції : (список_параметрів)=0;

Ключовою частиною цього оголошення є прирівнювання функції до нуля. Це повідомляє компілятору про те, що в базовому класі не існує тіла функції. Якщо клас містить хоча б одну чисто віртуальну функцію, то про нього говорять як про абстрактний клас. Такий клас не може створити жодного об'єкта, він може бути тільки успадкованим.

Як приклад, розглянемо програму, в якій використовується абстрактний клас AREA та два похідних класи RECTANGLE і TRIANGLE. В класі AREA об'явлена чисто віртуальна функція *getarea*(), яка повертає значення площі геометричної фігури. В похідному класі RECTANGLE вказана функція повертає значення площі прямокутника, а в класі TRIANGLE - значення площі трикутника.

```
#include <iostream.h>
class AREA
{
      float X,Y;   // розміри фігури
      public:
       AREA(float NewX, float NewY)
       {
        X=NewX;   Y=NewY;
       }
        void get_xy(float &X1, float &Y1)
       {    X1 = X;     Y1 = Y; }
       virtual float get_area()
       {
       cout << "Вам необхідно цю функцію перевизначити "
         return 0.0;
       }
};
class RECTANGLE : public AREA
{
       {
       float X1,Y1;
       public:
       RECTANGLE::RECTANGLE(float X2, float Y2):AREA(X2,Y2)
       {
         X1 = X2;   Y1 = Y2;
```

```
    }
    float get_area()
    {
     float X1, Y1;
     get_xy(X1,Y1);
     return X1 * Y1;
    }
    };
class TRIANGLE : public AREA
{
    float X1, Y1;
    public:
     TRIANGLE::TRIANGLE(float X2, float Y2):AREA(X2, Y2)
    {
     X1 = X2;   Y1 = Y2;
    }
    float get_area()
    {
     float X1, Y1;
     get_xy(X1, Y1);
     return 0.5 * X1 * Y1;
    }
};

void main()
{
     AREA *ptr;
    RECTANGLE rec(5.0, 4.5);
    TRIANGLE tri(3.0, 8.0);
    ptr = &rec;
     cout << " Площа прямокутника:" << ptr -> get_area() << '\n';
    ptr = &tri;
    cout << "Площа трикутника:" << ptr -> get_area() << '\n';
}.
```

4 Створення прикладних MFC-програм в середовищі Windows

Структура програм в середовищі Windows значно спрощується при використанні спеціальної бібліотеки Microsoft Foundation Classes (MFC). Перехід до бібліотеки MFC означає перехід до об'єктно-орієнтованого програмування мовою C++.

Бібліотека MFC побудована за ієрархічним принципом. На вершині ієрархії знаходиться єдиний клас CObject. Всі інші класи можна умовно розбити на дві категорії: похідні від класу CObject, і класи, що не залежать від CObject. Класи, які описують окремі компоненти Windows (вікна, діалоги, графіку тощо), як правило, створюють локальну ієрархію зі своїм базовим класом.

Кожен клас надає програмісту велику кількість різноманітних функцій. Оскільки в програмах можна використовувати як API-функції, так і функції класів, тому, щоб їх розрізнити, останні будемо називати MFC-функціями або методами.

4.1 Структура мінімальної прикладної MFC-програми

Перед тим як приступити до написання складних програм, розглянемо структуру найпростішої програми, завданням якої є лише виведення вікна програми на екран. В попередньому розділі ми реалізували цю задачу з допомогою *API*-функцій мовою С. Об'єм наступної програми буде в декілька раз менший.

Мінімальна *MFC*-програма містить два класи, які визначаються користувачем.

Перший клас *CMyApp*, є похідним від бібліотечного класу *CWinApp* і призначений для створення глобального об'єкта класу прикладних програм. Базовий клас *CWinApp* має всього два методи: конструктор і функцію ініціалізації *InitInstance*(). Перевизначаючи останню MFC-функцію, програміст одержує можливість керувати ініціалізацією прикладної програми.

Починається перевизначувана MFC-функція *InitInstance*() зі створення об'єкта фрейму вікна:

*CMyFrameWnd *pMainWnd=new CMyFrameWnd;*

потім із встановлення його як головного вікна прикладної програми

m_pMainWnd=pMainWnd; .

Далі вікно необхідно зробити видимим:

m_pMainWnd->ShowWindow(m_nCmdShow);

і поновити його вміст:

m_pMainWnd->UpdateWindow(); .

Дуже корисно зупинитися на особливостях позначень імен в бібліотеці *MFC*.

В якості префікса, який позначає ім'я класу, використовується заглавна буква *C* від слова *class*, за який йде власне ім'я класу. Наприклад *CWnd, CWinApp, CDialog* і інші. Такий же префікс рекомендується використовувати і для імен класу користувача.

Для членів класу прийнятий такий спосіб утворення імен: обов'язковий префікс *m_* (від слів *class member*), за яким йде префікс, який характеризує тип даних, а потім йде власне ім'я змінної, наприклад *m_pMainWnd*, де *p* – префікс покажчика. Для найменування змінних, які не є членами будь-якого класу, префікс *m_* не використовується.

Кожна *MFC*-програма буде практично завжди містити хоча б один клас, похідний від бібліотечного віконного класу. В наступній мінімальній *MFC*-програмі визначається власний клас *CMyFrameWnd*, як похідний від класу *CFrameWnd*.

На клас *CFrameWnd* покладається багато задач, зокрема:

- створення перекриваючих і висхідних вікон,
- створення меню,
- створення панелей інструментів і рядка стану,
- створення акселераторів команд,
- підтримка операції "перенести і відпустити" (*drag-and-drop*).

Якщо більш коротко визначити призначення класу *CFrameWnd* то необхідно сказати, що з його допомогою легко створювати фрейми вікон (*Window frames*). Фрейм представляє собою видиму рамку і стандартні елементи вікна (смуга заголовка, кнопки максимізації і мінімізації, меню, панель інструментів, рядок стану) (рис. 4.1).

В нашій мінімальній *MFC*-програмі конструктор класу *CMyFrameWnd* викликає метод *Create*(), призначений для встановлення початкових параметрів вікна.

Розглянуті вище функції і методи класів *CMyApp* і *CMyFrameWnd* відповідають за створення, відображення, і відновлення вікна, реєстрацію класу вікна, тобто виконує ті ж дії, що й функція *WinMain*() в мінімальній API-програмі для *Windows* на мовою *C*. Залишилось тільки розглянути особливості виконання *MFC*-програм.

Виконання цієї найважливішої задачі взяла на себе функція *Run*() класу *CMyApp*. Результатом виклику цієї функції і є запуск циклу обробки повідомлень і поява на екрані стандартного вікна *Windows*, точно такого ж, як і при виконанні мінімальної *API*-програми. Саме тому *MFC*-програма і закінчується викликом стандартної функції *Run*().

Нарешті, не забути на самому початку програми підключити бібліотеки *MFC* до програми:

#include <afxwin.h>.

Через файл *afxwin.h* підключаються всі інші заголовочні файли, включаючи *windows.h*.

В результаті отримаємо варіант мінімальної MFC-програми.

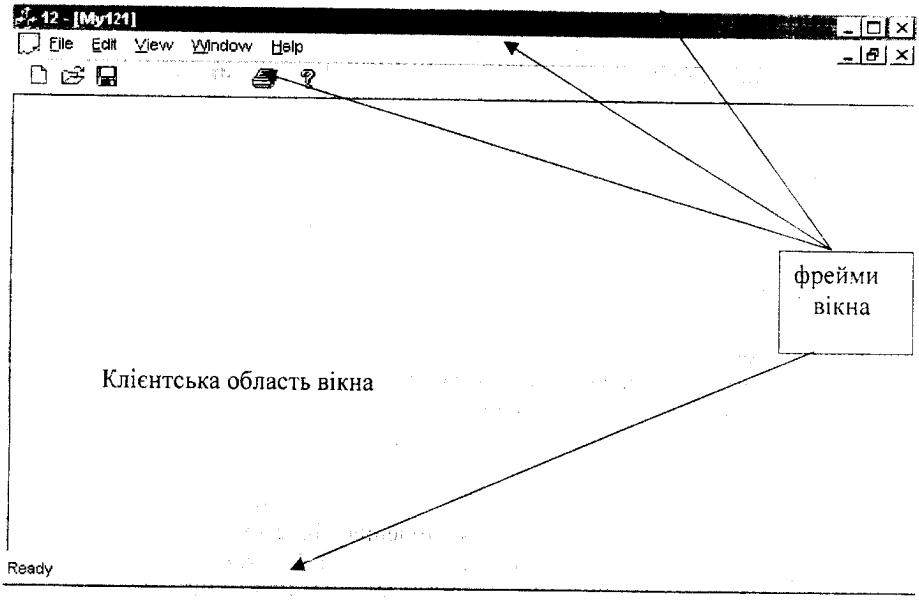


Рисунок 4.1 – Вікно MFC-програми

```
#include <afxwin.h>
class CMyFrameWnd:public CFrameWnd
{
  public:
        CMyFrameWnd();
  };

CMyFrameWnd::CMyFrameWnd()
{
        Create(NULL,"My first program",
        WS_OVERLAPPEDWINDOW,rectDefault,NULL,NULL);
};

class CMyApp:public CWinApp
{
 public:
        virtual BOOL InitInstance();
};
BOOL CMyApp::InitInstance()
{
        CMyFrameWnd *pMainWnd=new CMyFrameWnd;
        m_pMainWnd=pMainWnd;
        m_pMainWnd->ShowWindow(m_nCmdShow);
```

```
    m_pMainWnd->UpdateWindow();
    return TRUE;
};

CMyApp app;
```

4.2 Обробка повідомлень в MFC-програмах

В бібліотеці *MFC* всі повідомлення поділяються на три категорії:

- повідомлення *Windows*;
- повідомлення елементів керування;
- командні повідомлення (команди).

В першу категорію входять апаратні повідомлення (від миші і клавіатури), повідомлення обслуговування вікон, повідомлення про завершення роботи і багато інших. Всі повідомлення цієї категорії починаються з префікса *WM_*, за винятком *WM_COMMAND*.

Друга категорія включає повідомлення від елементів керування і інших дочірніх вікон, які направляються своїм батьківським вікнам. За незначними винятками, механізм передачі повідомлень аналогічний механізму передачі інших *WM_*повідомлень.

Повідомлення цих двох категорій призначені для об'єктів класів, утворених від бібліотечного класу *CWnd*.

Третя категорія включає всі повідомлення типу *WM_COMMAND* від об'єктів інтерфейсу користувача: меню, кнопок панелей інструментів і клавіш акселераторів. Механізм обробки командних повідомлень відрізняється від обробки інших повідомлень і може проводитися різними об'єктами, включаючи, наприклад, документи, і навіть самим об'єктом "програма".

Але незалежно від категорії повідомлення, в прикладній програмі вони повинні бути оброблені, тобто програма повинна певним чином відреагувати на надходження повідомлення. При використанні тільки *API*-функції *Windows* в прикладній програмі, як вже відмічалося в попередньому розділі, є спеціальна віконна функція, де і передбачені оброблювачі повідомлень. В *MFC*-програмах аналогом віконної функції виступає карта повідомлень (*MESSAGE_MAP*). Карта повідомлень перетворює всі категорії повідомлень в функції – обробники відповідних класів.

Розглянемо правила використання карти повідомлень.

По-перше, кожен клас, в якому повинна виконуватися обробка повідомлень, повинен мати власну карту повідомлень. При цьому слід пам'ятати, що вона не може визначатися всередині класу або функції. В класі (зазвичай в самому кінці), лише оголошується карта повідомлень за допомогою макросу *DECLARE_MESSAGE_MAP*.

Потім, за межами класу, визначається карта повідомлень таким чином. Починається вона макросом *BEGIN_MESSAGE_MAP*, якому в якості параметрів передається ім'я його "рідного" класу і безпосередньо базового класу. Закінчується карта повідомлень викликом макросу *END_MESSAGE_MAP*.

Між вказаними двома викликами розміщуються спеціальні макроси, які власне і зв'язують повідомлення з відповідною функцією – оброблювачем. Для кожної категорії повідомлень визначається свій тип макросу.

Для стандартних повідомлень *Windows* ім'я такого макросу має формат:

ON_WM_XXX // де XXX – ім'я каталога.

Наприклад, для обробки повідомлення від натиснення лівої клавіші миші, передбачений макрос

ON_WM_LBUTTONDOWN().

Для командних повідомлень формат імені такий:

ON_COMMAND (ID, Func),

де *ID* – ідентифікатор команди, визначений в ресурсних файлах (**.rc, Resource.h*).

Func – ім'я функції–оброблювача.

Прототип функції-оброблювача повинен бути описаний в класі, який є власником карти повідомлень, і має вигляд:

afx_msg voin Func(); .

Для оброблення повідомлень від елементів керування застосовується макрос

ON_CONTROL(Code, ID, Func),

де *Code* – код повідомлення від елемента керування;

ID – ідентифікатор елемента керування, визначений в ресурсних файлах;

Func – ім'я метода-оброблювача.

Крім розглянутих, можуть використовуватися і інші типи макросів, повний список яких можна знайти в файлі *afxmsg.h*.

4.3 Розробка меню в MFC-програмах

Процес створення меню в MFC-програмах, як і в API-програмах, складається із трьох етапів:

- створення шаблона меню;
- підключення ресурсу меню до головної програми;
- обробка повідомлень від команд меню.

Створення шаблона меню, візуально або програмно, в MFC-програмах відбувається аналогічно, як і в API-програмах. Відмінність починається з опису підключення ресурсу меню до головної програми (файла **.cpp*).

Існує декілька способів вказування ідентифікатора ресурсу меню в MFC-програмах.

4.3.1 Перший спосіб підключення меню в MFC-програмах

Перший спосіб базується на використанні методу *Create*() класу *CFrameWnd*, який створює головне вікно програми. В цьому методі можна вказати спеціальний макрос *MAKEINTRESOURCE*, аргументом якого буде ідентифікатор підключуваного ресурсу меню. В результаті отримаємо такий варіант мінімальної MFC-програми з підключеним меню:

```
#include <afxwin.h>
#include "Resource.h"
class CMyFrameWnd:public CFrameWnd
{
  public:
       CMyFrameWnd();
};

CMyFrameWnd::CMyFrameWnd()
{
       Create(NULL,"My Program",
        WS_OVERLAPPEDWINDOW,rectDefault,NULL,
              MAKEINTRESOURCE(IDR_MAINFRAME));
};

class CMyApp:public CWinApp
{
 public:
       virtual BOOL InitInstance();
};
BOOL CMyApp::InitInstance()
{
       CMyFrameWnd *pMainWnd=new CMyFrameWnd;
       m_pMainWnd=pMainWnd;
       m_pMainWnd->ShowWindow(m_nCmdShow);
       m_pMainWnd->UpdateWindow();
         return TRUE;
};
 CMyApp app;
```

Наведеній вище програмі відповідає такий ресурсний файл (файл типу *.rc):

```
#include "Resource.h"
IDR_MAINFRAME  MENU  DISCARDABLE
BEGIN
  POPUP "File"
  BEGIN
    MENUITEM "Exit",  IDM_EXIT
  END
    POPUP "Information"
  BEGIN
    MENUITEM "About...",  IDM_ABOUT
  END
END
```

Ідентифікатори меню та всіх пунктів меню мають бути представлені в заголовочному файлі ресурсів *Resource.h*:

```
#define IDR_MAINFRAME          128
#define IDM_ABOUT              105
#define IDM_EXIT               106
```

4.3.2 Другий спосіб підключення меню в MFC-програмах

Другий спосіб базується на використанні методу *LoadFrame*(), який викликається в методі *InitInstance*() класу, похідному від бібліотечного класу *CWinApp*.

Оскільки метод *LoadFrame*() викликає метод *CMyFrameWnd::Create*(), тому метод *Create*() непотрібно явно викликати в конструкторі класу вікна *CMyFrameWnd*. В результаті отримаємо ще один варіант мінімальної MFC-програми з підключеним меню:

```
#include <afxwin.h>
#include "Resource.h"
class CMyFrameWnd:public CFrameWnd
{
 public:
      CMyFrameWnd()
      {  }
};

class CMyApp:public CWinApp
{
 public:
      virtual BOOL InitInstance();
};
```

```
BOOL CMyApp::InitInstance()
{
      CMyFrameWnd *pMainWnd=new CMyFrameWnd;
      pMainWnd->LoadFrame(IDR_MAINFRAME);
      m_pMainWnd=pMainWnd;
      m_pMainWnd->ShowWindow(m_nCmdShow);
      m_pMainWnd->UpdateWindow();
        return TRUE;
};
 CMyApp app;.
```

4.3.3 Обробка повідомлень від команд меню

Після запуску на виконання останніх двох програм в верхній частині клієнтської області вікна можна буде побачити створене меню.

Тепер вже можна перейти до третього етапу створення меню. Обробку повідомлень від команд меню в головному програмному файлі проекту необхідно виконати в такій послідовності.

1. В класі вікна з рамкою *CMyFrameWnd* оголосити функції-обробники всіх пунктів меню.

2. Включити в карту повідомлень *MESSAGE_MAP* макроси, які зв'язують ідентифікатори кожного пункту меню з функцією-обробником.

3. Написати всі функції-обробники, які були вказані в класі вікна з рамкою *CMyFrameWnd* та карті повідомлень *MESSAGE_MAP*.

Найбільш поширеною реакцією на активізацію пунктів меню може бути поява на екрані вікна повідомлень *MessageBox*.

Як приклад розглянемо MFC-програму з двома активними пунктами меню. При активізації першого пункту меню *About* на екран видається вікно повідомлення *MessageBox*, а при активізації другого пункту меню *Exit* – закривається головне вікно і програма закінчує свою роботу.

```
#include <afxwin.h>
#include "Resource.h"
class CMyFrameWnd:public CFrameWnd
{
  public:
       CMyFrameWnd();
   protected:
    afx_msg void OnExit();
    afx_msg void OnAbout();
    DECLARE_MESSAGE_MAP();
 };
BEGIN_MESSAGE_MAP(CMyFrameWnd, CFrameWnd)
    ON_COMMAND(IDM_EXIT,OnExit)
```

```
    ON_COMMAND(IDM_ABOUT,OnAbout)
END_MESSAGE_MAP();
CMyFrameWnd::CMyFrameWnd()
{
        Create(NULL,"My Program",
         WS_OVERLAPPEDWINDOW,rectDefault,NULL,                    /
           MAKEINTRESOURCE(IDR_MAINFRAME));
};

void CMyFrameWnd::OnAbout()
 {
        ::MessageBox(NULL,"Copyright by Semerenko","About",MB_OK
              MB_ICONEXCLAMATION);
 }
 void CMyFrameWnd::OnExit()
 {
   SendMessage(WM_CLOSE);
 }

class CMyApp:public CWinApp
{
 public:
        virtual BOOL InitInstance();
};
BOOL CMyApp::InitInstance()
{
        CMyFrameWnd *pMainWnd=new CMyFrameWnd;
        m_pMainWnd=pMainWnd;
        m_pMainWnd->ShowWindow(m_nCmdShow);
        m_pMainWnd->UpdateWindow();
          return TRUE;
};
 CMyApp app;
```

Ресурсний файл (файл типу *.rc) та заголовочний файл ресурсів *Resource.h* для останньої програми такі ж, як і в п. 4.3.1.

Можна ввести ще ряд модифікацій в створене меню:

- ввести клавіші прискореного виклику пунктів меню;
- ввести акселератори, тобто сполучення клавіш, за допомогою яких можна вибирати пункти меню навіть в недоступних меню;
- забезпечити почергову активізацію та деактивізацію пунктів меню.

В Visual C++ 6.0 підтримуються також динамічні та контекстні меню. Ці різновидності меню є об'єктами бібліотечного класу *CMenu*.

4.4 Підключення панелі інструментів і рядка стану в MFC-програмах

Будь-яка сучасна професійна програма має в своєму головному вікні різні інструментальні засоби, які значно полегшують взаємодію з користувачем. До таких засобів, окрім традиційних меню, відносять також панель інструментів (*Toolbar*) і рядок стану (*Statusbar*).

Панель інструментів (toolbar) – це сукупність растрових кнопок одного розміру (стандарт – 23 · 22 пікселів) і розподілювачів. Натиснення на кнопку панелі інструментів подібно вибору одного пункту меню. По суті, панель інструментів дублює роботу меню, однак, на відміну від меню, вона більш зручна в роботі завдяки додатковим властивостям: організації підказок, встановленню різноманітних стилів і розмірів кнопок тощо.

Рядок стану – це рядок внизу робочої області батьківського вікна, який можна розділити на декілька областей для окремого виведення в ці області тексту або графічної інформації. Найчастіше рядок стану використовується для виведення інформації про елементи меню і кнопки панелі інструментів а також для відображення індикаторів стану (наприклад індикаторів *CAPSLOCK*, *NUMLOCK*, *SCROLLOCK*).

Процес створення панелі інструментів і рядка стану має дуже багато спільного і складається з двох головних етапів:

- створення ресурсу відповідного інструментального засобу;
- написання програмного коду.

Розглянемо спочатку послідовність візуального проектування панелі інструментів як елемента ресурсу проекту.

4.4.1 Створення панелі інструментів

Якщо натиснути правою кнопкою миші на папці *Toolbar* в вікні *Resource View*, а потім вибрати із контекстного меню *Insert Toolbar*, то буде створена нова (поки ще пуста) панель інструментів. При її створені необхідно задати відповідний ідентифікатор *ID*. Якщо раніше вже було створене меню, то ідентифікатор панелі інструментів повинен збігатися з ідентифікатором меню, наприклад *IDR_MENU*.

Далі панель інструментів заповнюється кнопками. *Visual C++* надає в розпорядження користувача графічний редактор з дуже великими можливостями. По-перше можна змінити розміри кнопки (за замовчуванням кнопка має розмір 16×15 пікселів), її колір і рисунок. Після того як кнопка "нарисована", їй необхідно присвоїти ідентифікатор. Оскільки панель інструментів, що створюється буде дублювати дію меню, тому кожній кнопці присвоюється ідентифікатор, що збігається з ідентифікатором відповідного пункту меню.

Після створення всіх необхідних кнопок панелі інструментів, можна використати також додаткові можливості візуального редактора: змінити

порядок розміщення кнопок, організувати їх в групи або вилучити. Можна додати підказку для кнопки, записавши її в поле "*Prompt*" в діалоговому вікні властивостей кнопок.

Дозволяється задання початкового стану кнопки: натиснута, заблокована, невизначена і не натиснута (стандартний стан). При цьому не потрібно створювати різні бітові зображення для кожного стану кнопки: *Windows* формує необхідне зображення автоматично, обробляючи лише один заданий стан кнопки.

Розглянемо тепер питання створення програмного коду, який необхідний для роботи з панеллю інструментів.

В бібліотеці *MFC* для панелей інструментів створені два класи:

- клас *CToolBar* – для визначення поведінки панелі інструментів;
- клас *CToolBarCtrl* – для визначення стандартного керуючого елемента *Windows* "Панель інструментів".

З позицій програмування панель інструментів – це спеціальне вікно, для створення якого необхідно виконати такі кроки.

1. В класі вікна з рамкою (як правило, похідному від *CFrameWnd*) необхідно створити об'єкт класу *CToolBar*:

CToolBar m_wndToolBar; .

2. Створити вікно Windows способом перевизначення функції *OnCreate*() класу вікна з рамкою. Вказане перевизначення полягає в доданні спеціальної API-функції, яка безпосередньо описує властивості панелі інструментів. В попередніх версіях Visual C++ для цієї мети використовувалась функція *Create*(), а в версії 6.0 – *CreateEx*(), яка може мати, наприклад, такий вигляд:

```
CToolBar.CreateEx(CWnd* pParentWnd,
        DWORD dwCtrlStyle=TBSTYLE_FLAT,
        DWORD dwStyle= WS_CHILD | WS_VISIBLE | CBRS_TOP
        CRect rcBorders=Crect(0,0,0,0)
        UINT nID=AFX_TOOLBAR);
```

Тут аргументами функції *CreateEx*() є такі:

pParentWnd - покажчик на батьківське вікно панелі інструментів;

dwCtrlStyle - стилі кнопок елемента керування, на якому базується панель інструментів;

dwStyle - стилі панелі інструментів;

rcBorders - розміри прямокутника, всередині якого розташована панель інструментів;

nID - ідентифікатор вікна панелі інструментів.

В табл.4.1 наведені стилі *dwStyle* панелі інструментів, які найчастіше використовуються в прикладних програмах.

Серед стилів *dwCtrlStyle* кнопок панелі інструментів найчастіше використовуються такі:

TBSTYLE_BOTTON – стандартна кнопка;

TBSTYLE_CHECK – кнопка з фіксацією після її натиснення; для повернення в початковий стан необхідне її повторне натискання;

TBSTYLE_CHECKGROUP – відмічається початок групи кнопок з фіксацією; кнопка залишається натиснутою, поки не буде натиснута інша кнопка з цієї групи;

TBSTYLE_WPARABLE – кнопки панелі інструментів можуть розташовуватись в кілька рядів при великій кількості кнопок.

3. Створити функції-обробники для кожної кнопки панелі інструментів.

Оскільки кнопки панелі інструментів в більшості випадків дублюють відповідні пункти меню, то функції-обробники можуть бути спільними для меню та панелі інструментів. Таке зв'язування пунктів меню та кнопок панелі інструментів відбувається ще на етапі візуального проектування панелі інструментів через їх однакові ідентифікатори.

Варто відмітити, що програмно можна також завантажити зображення кнопок панелі інструментів, якщо рисунок кнопок не був створений за допомогою графічного редактора. Для виконання цієї задачі можна використати метод *LoadBitmap*(), або метод *LoadToolBar*() класу *CToolBar*. Формати цих методів такі:

BOOL CToolBar::LoadBitmap(LPCTSTR lpszResourceName);

BOOL CToolBar::LoadBitmap(UINT nIDResource);

BOOL CToolBar::LoadToolBar(LPCTSTR lpszResourceName);

BOOL CToolBar::LoadToolBar(UINT nIDResource); .

Результатом виконання цих методів буде завантаження ресурсу зображення кнопок панелі інструментів, який задається або іменем - *lpszResourceName*, або ідентифікатором – *nIDResource*. Метод *LoadToolBar*() також автоматично зіставляє кнопки з командами.

Табл.4.1 - Стилі *dwStyle* панелі інструментів

Стиль вікна панелі інструментів	Значення
CBRS_TOP	Розташувати зверху робочої області батьківського вікна
CBRS_BOTTOM	Розташувати внизу робочої області батьківського вікна
CBRS_LEFT	Розташувати зліва робочої області батьківського вікна
CBRS_RIGHT	Розташувати справа робочої області батьківського вікна
CBRS_ANY	Дозволяється розташувати в будь-якій частині робочої області батьківського вікна
CBRS_TOOLTIPS	Виводити короткий опис кнопок

CBRS_SIZE_FIXED	Заборона зміни розмірів панелі інструментів
CBRS_FLOATING	Панель інструментів плаваюча
CBRS_FLYBY	В рядку стану відображати інформацію про кнопку
CBRS_HIDE_INPLACE	Зробити невидимою для користувача
CBRS_ALIGN_TOP	Панель інструментів прив'язується до верхньої сторони робочої області фрейму
CBRS_ALIGN_BOTOM	Панель інструментів прив'язується до нижньої сторони робочої області фрейму
CBRS_ALIGN_LEFT	Панель інструментів прив'язується до лівої сторони робочої області фрейму
CBRS_ALIGN_RIGHT	Панель інструментів прив'язується до правої сторони робочої області фрейму
CBRS_ALIGN_ANY	Панель інструментів прив'язується до будь-якої сторони робочої області фрейму

4.4.2 Створення рядка стану

Рядок стану може працювати в двох режимах – спрощеному і стандартному. В спрощеному режимі розподіл цього вікна на декілька областей неможливий і виводити на нього можна тільки текстову інформацію. В стандартному режимі прикладна програма може розбити вікно рядка стану на декілька областей для окремого виведення на них текстової і графічної інформації. Перехід із одного режиму в інший здійснюється за допомогою повідомлення *SB_SIMPLE* перед першим відображенням рядка на екрані.

Для створення рядка стану необхідно виконати такі кроки.

1. В класі вікна програми оголосити змінну класу *CStatusBar* для об'єкта – рядка стану:

CStatusBar m_wndStatusBar; .

2. Створити структуру з ідентифікаторами полів, що відображаються в рядку стану. Наприклад

```
static UINT indicators[] =
{
    ID_SEPARATOR,          // лінія розподілу
    ID_INDICATOR_CAPS,  // ідентифікатор клавіші CAPSLOCK
    ID_INDICATOR_NUM,  // ідентифікатор клавіші NUMLOCK
    ID_INDICATOR_SCRL,  // ідентифікатор клавіші SCROLLOCK
};
```

1. Перевизначити метод *OnCreate()* класу вікна програми і внести за допомогою макросу *ON_WM_CREATE* в карту повідомлень запис, який пов'язує метод з повідомленням *WM_CREATE* (якщо ці дії не були виконані раніше при створені панелі інструментів).

```
m_wndStatusBar.Create(this) ||
!m_wndStatusBar.SetIndicators(indicators,
sizeof(indicators)/sizeof(UINT));.
```

2. Викликати API-функцію *SetIndicators()*, якій передається структура з ідентифікатором полів, що відображаються в рядку стану:

```
m_wndStatusBar.Create(this);
m_wndStatusBar. SetIndicators(indicators, sizeof(indicators)/
sizeof (UNIT));.
```

4.4.3 Приклад програми

Як приклад, розглянемо повний опис класу фрейма вікна *CMyFrameWnd*, в якому підключається панель інструментів і рядок стану.

```
class CMyFrameWnd:public CFrameWnd
{
  public:
      CMyFrameWnd();
   protected:
      CStatusBar  m_wndStatusBar;
      CToolBar    m_wndToolBar;
      afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
      DECLARE_MESSAGE_MAP();
 };
BEGIN_MESSAGE_MAP(CMyFrameWin, CFrameWnd)
      ON_WM_CREATE()
END_MESSAGE_MAP();
static UINT indicators[] =
{
      ID_SEPARATOR,
      ID_INDICATOR_CAPS,
      ID_INDICATOR_NUM,
      ID_INDICATOR_SCRL,
};
CMyFrameWnd::CMyFrameWnd()
{
      Create(NULL,"My Program",
       WS_OVERLAPPEDWINDOW,rectDefault,NULL,
            MAKEINTRESOURCE(IDR_MAINFRAME));
};
int CMyFrameWnd::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
      if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
            return -1;
```

```
    if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD
    | WS_VISIBLE | CBRS_TOP | CBRS_GRIPPER | CBRS_TOOLTIPS
            | CBRS_FLYBY | CBRS_SIZE_DYNAMIC) ||
      !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
    {
        TRACE0("Failed to create toolbar\n");
        return -1;
    }
    if (!m_wndStatusBar.Create(this) ||
        !m_wndStatusBar.SetIndicators(indicators,
         sizeof(indicators)/sizeof(UINT)))
    {
        TRACE0("Failed to create status bar\n");
        return -1;
    }
    return 0;
}
```

4.5 Векторна графіка в MFC-програмах

4.5.1 Контексти пристроїв в MFC-програмах

Як вже раніше відмічалось, все виведення в системі Windows побудоване таким чином, щоб уніфікувати роботу з різними фізичними пристроями: екраном дисплея, принтером, плотером і т.д. Для всіх таких пристроїв графічного виведення використовуються одні і ті ж самі функції і класи, а вибір конкретного пристрою система виконує самостійно на основі інформації про контекст пристрою. Особливістю бібліотеки *MFC* є те, що вона містить спеціальні класи контекстів пристроїв. Базовим класом для всіх контекстів пристроїв є клас *CDC*. Існує також ще декілька класів контекстів пристроїв, що мають більш вузьку спеціалізацію (табл.4.2).

4.5.2 Використання пера та пензля в MFC-програмах

Перед початком виведення на пристрої графічних зображень необхідно також попередньо настроїти відповідні графічні об'єкти: перо, пензель, шрифти і т.д. В бібліотеці *MFC* кожному такому об'єкту відповідає свій клас (табл.4.3).

Послідовність створення графічних об'єктів на основі класів бібліотеки MFC дуже нагадує відповідну процедуру при роботі з функціями API.

1. Створення графічного об'єкта. Це можна зробити двома способами.

Перший спосіб. Спочатку викликається конструктор для створення об'єкта класу, а потім викликається відповідна MFC-функція ініціалізації.

Наприклад:

CPen Pen1; // створення пера;

Pen1.CreatePen(PS_SOLID,2,RGB(255, 0, 0)); // ініціалізація червоного пера для рисування суцільних ліній товщиною в два пікселя;

Таблиця 4.2 - Класи контекстів пристроїв

Клас	*Призначення*
CDC	Базовий клас усіх класів контекстів пристроїв
CClientDC	Клас контексту пристрою для забезпечення доступу до клієнтської (робочої) області вікна. При створенні об'єкта цього класу в конструкторі викликається API -функція *GetDC()*, а при його знищенні в деструкторі – API-функція *RealeaseDC()*.
CWindowDC	Клас контексту пристрою для забезпечення доступу до всього вікна. При створенні об'єкта цього класу в конструкторі викликається API-функція *GetWindowDC()*, а при його знищенні в деструкторі – API-функція *ReleaseDC()*.
CPaintDC	Клас контексту пристрою для використання тільки в обробнику повідомлення *WM_PAINT* з метою збереження і відновлення вмісту вікна. При створенні об'єкта цього класу в конструкторі викликається API-функція *BeginPaint()*, а при його знищенні в деструкторі – API-функція *EndPaint()*.
CMetaFileDC	Клас контексту пристрою для забезпечення доступу до метафайлів. Метафайли (тип файлів *.WMF* або *.EMF*), на відміну від інших графічних файлів, містять команди рисування, які виконуються при виведенні метафайла.

Другий спосіб. Створення графічного об'єкта і його одночасна ініціалізація з допомогою одного конструктора.

Наприклад:

CPen Pen1(PS_SOLID , 2, RGB(255, 0, 0));

Таблиця 4.3 - Класи графічних об'єктів

Клас	Тип дескриптора	Призначення
CPen	*HPEN*	Призначений для створення об'єктів "перо"
CBrush	*HBRUSH*	Призначений для створення об'єктів "пензель"
CFont	*HFONT*	Призначений для створення об'єктів "шрифт"
CBitmap	*HBITMAP*	Призначений для створення об'єктів "бітовий масив"
CPalette	*HPALETTE*	Призначений для створення об'єктів "палітра", тобто набір кольорів, доступних системі
CRgn	*HRGN*	Призначений для створення об'єктів – "областей" різних форм

2. Завантаження графічного об'єкта в контекст пристрою. Виконується з допомогою методу *SelectObject*(). Наприклад:

CClient dc(this);

dc.SelectObject(Pen1); .

3. Після закінчення операції рисування забезпечити знищення графічного об'єкта з контексту пристрою. Ця операція в *MFC*-програмах відбувається автоматично. Нагадаємо, що при роботі з API-функціями знищення графічного об'єкта проводиться в явному вигляді за допомогою функції *DeleteObject*().

4.5.3 Рисування ліній

Методи класів для рисування ліній та геометричних фігур і за назвами і за виконуваними функціями дуже схожі на відповідні функції API. Тому лише коротко зупинимось на найважливіших методах векторної графіки.

Рисувати будемо в контексті робочої області вікна.

Щоб нарисувати одиночну пряму лінію, спочатку необхідно вказати її початкову точку з координатами *x, y*, за допомогою методу *MoveTo*:

BOOL MoveTo(int x, int y); .

Потім необхідно викликати метод *LineTo*:

- *BOOL LineTo(int* x_1*, int* y_1*)*; .

Цей метод рисує пряму від поточної позиції до точки з координатами x_1 і y_1. Подальші виклики цього методу дозволять нарисувати задану послідовність ліній

Таку ж послідовність ліній, які з'єднуються між собою, можна нарисувати з допомогою методу *PolyLine*:

*BOOL PolyLine(CONST POINT *lppt, int cPoints)*;

де *lppt* представляють собою масив структур типу *POINT*, який містить координати набору точок, а *cPoints* – кількість самих точок (яких повинно бути не менше двох). Наприклад, з допомогою наступного фрагмента програми можна нарисувати букву М:

```
POINT Points[5];
Points[0].x=50; Points[0].y=50;
Points[1].x=50; Points[1].y=150;
Points[2].x=100; Points[2].y=100;
Points[3].x=150; Points[3].y=50;
Points[4].x=150; Points[4].y=150;
dc.PolyLine(Points, 5);.
```

Нарисувати дугу можна з допомогою методу *Arc*:

BOOL RoundRect(int x_1, int y_1, int x_2, int y_2, int x_3, int y_3, int x_4, int y_4); .

Призначення аргументів в останньому методі такі ж, як і в однойменній функції API.

4.5.4 Рисування стандартних геометричних фігур

Для побудови прямокутника використовується метод *Rectangle*:

BOOL Rectangle(int x_1, int y_1, int x_2, int y_2): ,

а для побудови еліпса – метод *Ellipse*:

BOOL Ellipse(HDC hdc, int x_1, int y_1, int x_2, int y_2); .

Тут параметри x_1 і y_1 визначають координати лівого верхнього кута прямокутника, а параметри x_2 і y_2 – координати нижнього правого кута прямокутника (рис.2.2, рис. 2.3).

Для рисування різноманітних багатокутників можна використати метод *Polygon*:

*BOOL Polygon(CONST POINT *lpPoints, int nCount)*; .

Параметр *lpPoints* представляє собою масив структур *POINT*, які містять координати вершин, а *nCount* є числом точок в масиві (їх повинно бути не менше двох).

В MFC-програмах можна також нарисувати сегмент за допомогою методу *Chord*:

BOOL Chord(HDC hdc, int x_1, int y_1, int x_2, int y_2, int x_3, int y_3,
int x_4, int y_4); .

а також сектор за допомогою методу *Pie*:

BOOL Pie(HDC hdc, int x_1, int y_1, int x_2, int y_2, int x_3, int y_3,
int x_4, int y_4); .

Призначення аргументів в останніх двох методах такі ж, як і в однойменних функціях API (рис.2.4, рис. 2.5).

4.5.5 Приклад програми з векторною графікою

Як приклад розглянемо програму, що формує зображення трьох геометричних фігур: прямокутника, еліпса, трикутника та текстові підказки. Зображення кожної фігури виводиться в робочу область вікна програми при виборі відповідного пункту меню. При заданні розмірів прямокутника і еліпса використовується клас CRect.

```
#include <afxwin.h>
#include "Resource.h"
class CMyFrameWnd:public CFrameWnd
{
  public:
       CMyFrameWnd();
   protected:
    afx_msg void OnRectangle();
    afx_msg void OnEllipse();
    afx_msg void OnPoligon();
    afx_msg void OnExit();
    afx_msg void OnAbout();
    DECLARE_MESSAGE_MAP();
 };

BEGIN_MESSAGE_MAP(CMyFrameWnd, CFrameWnd)
  ON_COMMAND(IDM_RECTANGLE,OnRectangle)
  ON_COMMAND(IDM_ELLIPSE,OnEllipse)
  ON_COMMAND(IDM_POLIGON,OnPoligon)
  ON_COMMAND(IDM_EXIT,OnExit)
  ON_COMMAND(IDM_ABOUT,OnAbout)
END_MESSAGE_MAP();

CMyFrameWnd::CMyFrameWnd()
{
       Create(NULL,"My 4th program",
        WS_OVERLAPPEDWINDOW,rectDefault,NULL,
              MAKEINTRESOURCE(IDR_MAINFRAME));
};
void CMyFrameWnd::OnRectangle()
 {
       CClientDC dc(this);
       CPen   pen;
```

```
        CBrush brush;
        CRect rect(30,50,250,200);
    pen.CreatePen(PS_SOLID, 3, RGB(0,0,255));
        brush.CreateSolidBrush(RGB(0,255,0));
    dc.SelectObject(&pen);
        dc.SelectObject(&brush);
        dc.FillRect(&rect,&brush);
        dc.Rectangle(&rect);
        dc.DrawText("It is Rectangle",rect,3);
        pen.DeleteObject();
        brush.DeleteObject();
  }
void CMyFrameWnd::OnEllipse()
  {
        CClientDC dc(this);
        CPen  pen;
    CBrush brush;
        CRect rect;
        GetClientRect(&rect);
    pen.CreatePen( PS_SOLID, 5, RGB( 0, 127, 127));
    dc.SelectObject(&pen);
     brush.CreateHatchBrush(HS_CROSS, RGB( 255, 0, 0));
    dc.SelectObject(&brush);
    dc.Ellipse(rect.left+300, rect.top+50,
                rect.right-120,rect.bottom-200);
    dc.SetTextColor( RGB( 100, 0, 255));
    dc.TextOut(350,110,"It is Ellipse");
    pen.DeleteObject();
    brush.DeleteObject();
  }
void CMyFrameWnd::OnPoligon()
  {
    CClientDC dc(this);
    CPen  pen;
    CBrush brush;
    CPoint mas[3];
    mas[0].x=175;
    mas[0].y=350;
    mas[1].x=350;
    mas[1].y=175;
    mas[2].x=275;
    mas[2].y=350;
    pen.CreatePen( PS_SOLID, 3, RGB( 0, 255 , 0));
    dc.SelectObject( &pen );
```

```
    brush.CreateSolidBrush( RGB( 220, 0, 200));
    dc.SelectObject( &brush );
    dc.Polygon(mas,3);
    dc.SetTextColor( RGB( 200, 200, 100));
    dc.TextOut( 200 , 340, "It is Triangle");
    pen.DeleteObject();
    brush.DeleteObject();
  }
void CMyFrameWnd::OnAbout()
  {
        ::MessageBox(NULL,"Copyright by Semerenko","About",MB_OK|
              MB_ICONEXCLAMATION);
  }
 void CMyFrameWnd::OnExit()
  {
   SendMessage(WM_CLOSE);
  }

class CMyApp:public CWinApp
{
 public:
      virtual BOOL InitInstance();
};
BOOL CMyApp::InitInstance()
{
      CMyFrameWnd *pMainWnd=new CMyFrameWnd;
      m_pMainWnd=pMainWnd;
      m_pMainWnd->ShowWindow(m_nCmdShow);
      m_pMainWnd->UpdateWindow();
        return TRUE;
};

 CMyApp app;
```

4.6 Растрова графіка в MFC-програмах

Бітові зображення (БЗ) мають такі принципові відмінності від векторних зображень (ВЗ).

1. БЗ є самостійним ресурсом, яке програма використовує як єдине ціле.

2. БЗ створюється до початку виконання програми за допомогою вбудованого або зовнішнього графічного редактора. Можливе також створення БЗ за допомогою сканера, цифрової фото- або відеокамери.

3. БЗ завжди представляє собою прямокутник, тому достатньо лише вказати розмірність БЗ як прямокутника та послідовно запам'ятати номери точок.

4. БЗ є апаратно залежною графічною фігурою, тому якість зображення повністю визначається розподільною здатністю графічного пристрою виведення.

5. Основним елементом БЗ є точка (піксел в дисплеї), а у ВЗ основний елемент – лінія.

Для використання БЗ в прикладних програмах необхідно виконати дві основні дії:

- створити БЗ за допомогою графічного редактора або відповідних апаратних засобів;
- написати програмний код для підключення БЗ до програми.

Створення БЗ за допомогою вбудованого або зовнішнього графічного редактора з пакету Visual C++ відбувається аналогічним чином, як і при створення будь-якого іншого ресурсу (меню чи панелі інструментів). Розпочинаючи із головного меню пакета Visual C++, необхідно послідовно виконати команди Insert/Resource/Bitmap/New. Далі в діалоговому вікні графічного редактора треба задати параметри БЗ: ідентифікатор БЗ, розміри прямокутника БЗ, кількість використаних кольорів та ім'я файла для збереження БЗ на диску (тип файла - *. bmp). Нарешті, виконати основне – нарисувати саме зображення і зберегти його на диску. Варто відмітити, що ресурсний файл (файл *.rc) проекту буде містити не саме БЗ, а лише вказане ім'я файлу із БЗ.

Програмний код в головному файлі проекту (тип файлу *.cpp) повинен реалізувати схему передачі БЗ із дискового файла через оперативну пам'ять на пристрій графічного виведення, наприклад дисплей. З цією метою необхідно виконати такі дії.

1. Створити контекст оперативної пам'яті:

CDC memoryDC;

2. Створити контекст пристрою виведення, наприклад, робочої області екрана дисплея:

CClientDC dc(this); .

3. Створити об'єкт бібліотечного класу:

CBitmap bitmap; .

4. Вибрати БЗ із дискового файлу:

bitmap.LoadBitmap(IDB_BITMAP); .

5. Визначити розмір БЗ:

BITMAP Bm; .

bitmap.GetObject(sizeof(Bm),&Bm); .

6. Зв'язати контексти оперативної пам'яті та пристрою виведення:

memoryDC.CreateCompatibleDC(&dc); .

7. Завантажити БЗ в контекст оперативної пам'яті:

memoryDC.SelectObject(bitmap); .

7. Завантажити БЗ в контекст робочої області екрана дисплея, тобто відобразити БЗ на екрані дисплея:

dc.BitBlt(X,Y,Bm.bmWidth,Bm.bmHeight,&memoryDC,0,0,SRCCOPY);.

Таким чином, для відображення в робочій області екрана дисплея, створеного в графічному редакторі пакета Visual C++ бітового зображення, верхній лівий кут якого має координати *X=70, Y=100*, необхідно написати таку програму:

```
CDC memoryDC; CClientDC dc(this);
CBitmap bitmap;
BITMAP Bm;
      int X,Y;  X=70;    Y=100;
bitmap.LoadBitmap(IDB_BITMAP);
      bitmap.GetObject(sizeof(Bm),&Bm);
      memoryDC.CreateCompatibleDC(&dc);
memoryDC.SelectObject(bitmap);
dc.BitBlt(X,Y,Bm.bmWidth,Bm.bmHeight,&memoryDC,0,0,SRCCOPY);
```

Якщо БЗ знаходиться в окремому файлі типу *.bmp (наприклад в файлі *Figure.bmp*), тоді БЗ вибирається з дискового файла методом *LoadBitmap*:

```
      bitmap.LoadBitmap("Figure.bmp");
```

Можливі декілька способів виведення БЗ на екран дисплея, використовуючи різні варіанти останнього аргументу функції *BitBlt*:

SRCCOPY – копіювання БЗ із стиранням попереднього зображення; *SRCAND* – копіювання БЗ із виконанням побітової операції AND із попереднім зображенням; *SRCPAINT* - копіювання БЗ із виконанням побітової операції OR із попереднім зображенням; *SRCINVERT* - копіювання БЗ із виконанням побітової операції XOR із попереднім зображенням.

Для реалізації динамічних картинок необхідно імітувати рух БЗ по екрану за домомогою зміни координат X або Y.

```
for(int i=0;i<450;i++)
      {
dc.BitBlt(X,Y,Bm.bmWidth,Bm.bmHeight,&memoryDC,0,0,SRCCOPY);
   X++;
   for(int j=0;j<50000;j++);
      }.
```

4.7 Програмування діалогу в MFC-програмах

4.7.1 Основні етапи створення діалогу в MFC-програмах

Послідовність розробки діалогових програм на основі бібліотеки MFC така ж, як і послідовність розробки діалогових програм з використанням функцій API. Оскільки відмінність полягає лише у різних програмних кодах у файлі *.cpp, тому розглянемо цей файл детально.

Опис стандартного діалогового вікна в файлі *.cpp включає такі етапи.

1.Створення власного класу діалогу, похідного від бібліотечного класу *CDialog*.

В цьому класі мають бути оголошені необхідні змінні-властивості класу діалогу, які відповідають різним ЕК діалогового вікна. Ці змінні можуть бути двох категорій: значення (Value) та керування (Control). Змінні категорії Value можуть мати стандартні типи (*int, Cstring, BOOL* та ін.), тоді як змінні категорії Control визначаються як об'єкти класів відповідних ЕК. Вибір однієї з двох категорій визначає особливості реалізації наступних етапів.

2. Ініціалізація ЕК діалогового вікна.

Суть цього етапу полягає в присвоєнні змінним-властивостям тих значень, які мають бути відображені у відповідних ЕК при першому відображенні діалогового вікна. Ініціалізація ЕК відбувається, як правило, за допомогою стандартного методу *OnInitDialog*(). В окремих випадках, зокрема для змінних категорії Value, ініціалізація може виконуватись в конструкторі власного класу діалогу.

3. Організація обміну даними з ЕК діалогового вікна.

Найбільш зручним способом обміну даними є стандартний метод *DoDateExchange*(). Цей метод містить набір DDX-макросів, які встановлюють зв'язок між ЕК та відповідною змінною-властивістю класу діалогу. В таблиці 4.4 наведені DDX-макроси для найбільш поширених ЕК. Можуть також бути задані і DDV-макросів для контролю введених значень. Метод *DoDateExchange*() викликається автоматично перед відображенням діалогового вікна на екрані.Організувати обмін даними можна не тільки за допомогою методу *DoDateExchange*(), але також з використанням інших методів класів бібліотеки MFC.

4. Створення об'єкта власного класу діалогу та його активація.

Вказаний об'єкт створюється або методом *DoModal*() (для створення модального діалогу), або методом *Create*() (для створення немодального діалогу). Результатом цього етапу буде створення та відображення на екрані діалогового вікна.

5. Обробка введених користувачем даних в діалоговому вікні.

Виконується така обробка після натиснення кнопки *Ok* та закриття діалогового вікна. Після цього відбувається передача даних із ЕК у змінні-властивості класу діалогу згідно з DDX-макросами методу *DoDateExchange*(). Якщо введені дані не потрібно зберігати або обробляти, тоді діалогове вікно закривається натисненням кнопки *Cancel*.

6. Використання результатів введених даних.

Введені за допомогою діалогового вікна дані далі можуть бути використані в функціях інших класів програми.

Таблиця 4.4 - DDX-макроси для найбільш поширених ЕК

Макрос	Тип даних	Опис
DDX_Text	*Cstring, UINT, long, float*	Обмін даними з полями введення (Edit)
DDX_Check	*int*	Стан ЕК "прапорець" (CheckBox)
DDX_Radio	*int*	Номер вибраної кнопки-перемикача (RadioButton)
DDX_LBString	*Cstring*	Рядок, вибраний в полі списку (ListBox)
DDX_CBString	*Cstring*	Рядок, вибраний в полі комбінованого списку (ComboBox)
DDX)LBIndex	*Int*	Номер запису, вибраного у списку (ListBox)
DDX_CBString	*int*	Номер запису, вибраного у комбінованому списку (ComboBox)
DDX_Control	*клас*	Покажчик на об'єкт класу ЕК

Далі розглянемо на прикладах організацію обміну даними з найбільш поширеними ЕК діалогових вікон.

4.7.2 Організацію обміну даними з полем введення *Edit*

Розглянемо введення однорозрядного числа з використанням змінної-властивості *m_edit* категорії Control. Після першого відображення діалогового вікна на екрані поле введення *Edit* має бути ініціалізовано текстовим рядком "Text". Далі користувач замість цього рядка вводить однорозрядне число, натискає кнопку *Ok* та закриває діалогове вікно. Результатом обробки введеного числа повинно бути його подвоєння.

1.Створення власного класу діалогу, похідного від бібліотечного класу *CDialog*.

```
class CMyDialog : public CDialog
{
public:
```

```
        CMyDialog(CWnd* pParent = NULL);
            enum { IDD = IDD_DIALOG1 };
        CEdit  m_edit;
protected:
        virtual void DoDataExchange(CDataExchange* pDX);
protected:
        virtual void OnOK();
        virtual void OnCancel();
        virtual BOOL OnInitDialog();
        DECLARE_MESSAGE_MAP()
        };.
```

2. Ініціалізація ЕК діалогового вікна.

```
BOOL CMyDialog::OnInitDialog()
{
        CDialog::OnInitDialog();
        m_edit.SetWindowText(CString("Text"));
        return TRUE;
        }.
```

3. Організація обміну даними з ЕК діалогового вікна.

```
void CMyDialog::DoDataExchange(CDataExchange* pDX)
{
        CDialog::DoDataExchange(pDX);
        DDX_Control(pDX, IDC_EDIT1, m_edit);
}.
```

4. Створення об'єкта власного класу діалогу та його активація.

```
        CMyDialog dialog;
        dialog.DoModal();.
```

5. Обробка введених користувачем даних в діалоговому вікні.

```
void CMyDialog::OnOK()
{
        char str[80];CString tmp;
        tmp="";
          m_edit.GetWindowText(str,80);
          tmp=tmp+str[0];
        CDialog::OnOK();
}.
```

6. Використання результатів введених даних.

```
         int y;
      y=atoi(tmp)*2;.
```

Розглянемо введення однорозрядного числа з використанням змінної-властивості *m_edit* категорії Value. Всі інші умови такі ж, як у попередньому прикладі.

1.Створення власного класу діалогу, похідного від бібліотечного класу *CDialog*.

```
class CMyDialog : public CDialog
{
public:
        CMyDialog(CWnd* pParent = NULL);
              enum { IDD = IDD_DIALOG1 };
        CString      m_edit;
protected:
        virtual void DoDataExchange(CDataExchange* pDX);
protected:
        virtual void OnOK();
        virtual void OnCancel();
        virtual BOOL OnInitDialog();
};
```

2. Ініціалізація ЕК діалогового вікна.

```
BOOL CMyDialog::OnInitDialog()
{
        CDialog::OnInitDialog();
        m_edit="Text";
        UpdateData(FALSE);
        return TRUE;
}
```

3. Організація обміну даними з ЕК діалогового вікна.

```
void CMyDialog::DoDataExchange(CDataExchange* pDX)
{
        CDialog::DoDataExchange(pDX);
        DDX_Text(pDX, IDC_EDIT1, m_edit);
        DDV_MaxChars(pDX, m_edit, 255);
}
```

4. Створення об'єкту власного класу діалогу та його активація.

```
   CMyDialog dialog;
   dialog.DoModal();
```

5. Обробка введених користувачем даних в діалоговому вікні.

```
void CMyDialog::OnOK()
{
        CString str; CString tmp;     tmp="";
        tmp=tmp+m_edit[0];
        UpdateData(TRUE);
        CDialog::OnOK();
}
```

6. Використання результатів введених даних.

```
    int y;
    y=atoi(tmp)*2;
```

Тепер наведемо текст програми повністю.

```
#include <afxwin.h>
#include <stdio.h>
#include <iostream.h>
#include <fstream.h>
#include <afxdlgs.h>
#include  "Resource.h"
CString str[80];
char buf[80];
int Number;
int type;
int N;
CString tmp;
class CMyDialog : public CDialog
{
public:
        CMyDialog(CWnd* pParent = NULL);
                enum { IDD = IDD_MATRIXDIALOG };
        CListBox m_size;
        int     m_datatype;
        CEdit  m_matrix;
protected:
        virtual void DoDataExchange(CDataExchange* pDX);
protected:
        virtual void OnOK();
        virtual void OnCancel();
        virtual BOOL OnInitDialog();
        DECLARE_MESSAGE_MAP()
};
CMyDialog::CMyDialog(CWnd* pParent)
        : CDialog(CMyDialog::IDD, pParent)
{
        m_datatype = 1;
}
void CMyDialog::DoDataExchange(CDataExchange* pDX)
{
        CDialog::DoDataExchange(pDX);
        DDX_Control(pDX, IDC_MATRIXSIZE, m_size);
        DDX_Radio(pDX, IDC_BYTE, m_datatype);
        DDX_Control(pDX, IDC_TEXTMATRIX, m_matrix);
}
```

```
BEGIN_MESSAGE_MAP(CMyDialog, CDialog)
END_MESSAGE_MAP()
void CMyDialog::OnOK()
{
        char str[80];
        tmp="";
        Number=m_size.GetCurSel();
        m_size.GetText(Number,str);
          m_matrix.GetWindowText(str,80);
          tmp=tmp+str[0];
               MessageBox(str,"Result");
        CDialog::OnOK();
}
void CMyDialog::OnCancel()
{
        CDialog::OnCancel();
}
BOOL CMyDialog::OnInitDialog()
{
        CDialog::OnInitDialog();
        int i;
        char str[10];
        for(i=2; i<11; i++)
        {
              wsprintf(str,"%d",i);
              m_size.AddString(str);
        }
        m_size.SetCurSel(1);
        m_matrix.SetWindowText(CString("Text"));
        return TRUE;
        }
class CMyFrameWnd:public CFrameWnd
{
public:
              CMyFrameWnd();
protected:
        afx_msg void OnInput();
        afx_msg void OnRun();
        afx_msg void OnResult();
        afx_msg void OnQuit();
        afx_msg void OnAbout();
        afx_msg void OnHelp();
        DECLARE_MESSAGE_MAP();
};
```

```
BEGIN_MESSAGE_MAP(CMyFrameWnd,CFrameWnd)
        ON_COMMAND(ID_CALC_INPUT,OnInput)
        ON_COMMAND(ID_CALC_RUN,OnRun)
        ON_COMMAND(ID_CALC_RESULT,OnResult)
        ON_COMMAND(ID_FILE_QUIT,OnQuit)
        ON_COMMAND(ID_INFO,OnAbout)
        ON_COMMAND(ID_HELP,OnHelp)
        END_MESSAGE_MAP()
CMyFrameWnd::CMyFrameWnd()
{
    Create(NULL,"program of kurs
work",WS_OVERLAPPEDWINDOW,rectDefault,
                NULL,MAKEINTRESOURCE(IDR_MENU1));
};
void CMyFrameWnd::OnInput()
{
        CMyDialog dialog;
        if(dialog.DoModal()==IDOK)
                type=dialog.m_datatype;
}
void CMyFrameWnd::OnRun()
{
        char s[80];  int y;
      y=atoi(tmp)*2;
                wsprintf(s,"%d",y);
                MessageBox(s,"Result");
}
void CMyFrameWnd::OnResult()
{
        char s[80];
        CClientDC dc(this);
        wsprintf(s,"Number is %d",Number+1);
        dc.TextOut(100,200,s);
        wsprintf(s,"Type is %d",type);
        dc.TextOut(200,200,s);
}
void CMyFrameWnd::OnQuit()
{
        exit(0);
}

void CMyFrameWnd::OnAbout()
{
        ::MessageBox(NULL,"Copyright by Semerenko","About",MB_OK
```

```
            MB_ICONEXCLAMATION);
}
void CMyFrameWnd::OnHelp()
{
 ::MessageBox(NULL,"Copyright by Semerenko","Help",MB_OK
            MB_ICONEXCLAMATION);
}

class CMyApp:public CWinApp
{
public:
      virtual BOOL InitInstance();
};
BOOL CMyApp::InitInstance()
{
                        CMyFrameWnd *pMainWnd=new CMyFrameWnd;
                        m_pMainWnd=pMainWnd;
                        m_pMainWnd->ShowWindow(SW_SHOW);

                        m_pMainWnd->UpdateWindow();
                        SetDialogBkColor(RGB(0,0,255));
                        return TRUE;
};

CMyApp app;
```

4.7.3 Організація обміну даними зі списком *ListBox* і комбінованим списком *ComboBox*

Розглянемо введення однорозрядного числа з використанням змінних-властивостей *m_size* для списка та *m_combo* для комбінованого списку категорії Control. Після першого відображення діалогового вікна на екрані обидва списки мають бути ініціалізовані числами від 2 до 6, причому спочатку має бути показаним другий елемент списку. Далі користувач вибирає один елемент списку, натискає кнопку Ok та закриває діалогове вікно. Результатом обробки вибраного числа повинно бути його виведення через вікно повідомлення *MessageBox*.

1.Створення власного класу діалогу, похідного від бібліотечного класу *CDialog*.

```
class CMyDialog : public CDialog
{
public:
      CMyDialog(CWnd* pParent = NULL);
```

```
        enum { IDD = IDD_MATRIXDIALOG };
    CListBox m_size;
    CComboBox m_combo;
protected:
    virtual void DoDataExchange(CDataExchange* pDX);
protected:
    virtual void OnOK();
    virtual void OnCancel();
    virtual BOOL OnInitDialog();
    DECLARE_MESSAGE_MAP()
};
```

2. Ініціалізація ЕК діалогового вікна.

```
BOOL CMyDialog::OnInitDialog()
{
    CDialog::OnInitDialog();
    int i;
    char str[10];
    m_combo.AddString("2");
    m_combo.AddString("3");
    m_combo.AddString("4");
    m_combo.AddString("5");
    m_combo.AddString("6");
    m_combo.SetCurSel(1);

    for(i=2; i<7; i++)
    {
        wsprintf(str,"%d",i);
        m_size.AddString(str);
    }
    m_size.SetCurSel(1);
    UpdateData(FALSE);
    return TRUE;
}
```

3. Організація обміну даними з ЕК діалогового вікна.

```
void CMyDialog::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    DDX_Control(pDX, IDC_LISTBOX1, m_size);
    DDX_Control(pDX, IDC_COMBO1, m_combo);
}
```

4. Створення об'єкта власного класу діалогу та його активація.

```
CMyDialog dialog;
dialog.DoModal();
```

5. Обробка введених користувачем даних в діалоговому вікні.

```
void CMyDialog::OnOK()
{
    m_combo.GetWindowText(s,80);

    Number=m_size.GetCurSel();
    m_size.GetText(Number,str);
    UpdateData(TRUE);
        CDialog::OnOK();
}.
```

6. Використання результатів введених даних.

```
MessageBox(str,"ListBox");
MessageBox(s,"ComboBox");.
```

4.8 Використання DLL-бібліотек в MFC-програмах

4.8.1 Призначення бібліотек DLL

На відміну від виконувальних файлів формату EXE, динамічно підключувані бібліотеки (DLL), як правило, безпосередньо не виконуються і повідомлень не обробляють. Призначення бібліотек DLL – бути викликаними процесами (тобто програмами в стадії виконання) або іншими бібліотеками в потрібний момент. Ці бібліотеки беруть участь в роботі лише тоді, коли викликається одна з її функцій.

Саме тому більшість системних файлів Windows, які мають бути доступними для всіх прикладних та системних процесів, реалізовані у форматі DLL: KERNEL32.DLL, GDI32.DLL, USER32.DLL та інші файли. Необхідно також відмітити, що динамічні бібліотеки не обов'язково містять програмний код. Наприклад, файли шрифтів (типу *.FON) також є динамічними бібліотеками, які містять лише один ресурс – опис шрифтів.

Отже, бібліотеки DLL – це програмний код або дані, які динамічно доступні всім процесам операційної системи.

Кожний процес має свій окремий адресний простір. Коли виникає необхідність роботи з якою-небудь бібліотекою DLL, процес копіює її в свій адресний простір. Однак сама бібліотека DLL залишається в одному й тому ж місці фізичної пам'яті. В значній економії пам'яті полягає найважливіша перевага динамічних бібліотек.

4.8.2 Створення власної бібліотеки DLL

Для створення власної бібліотеки DLL спочатку необхідно створити відповідний проект таким чином.

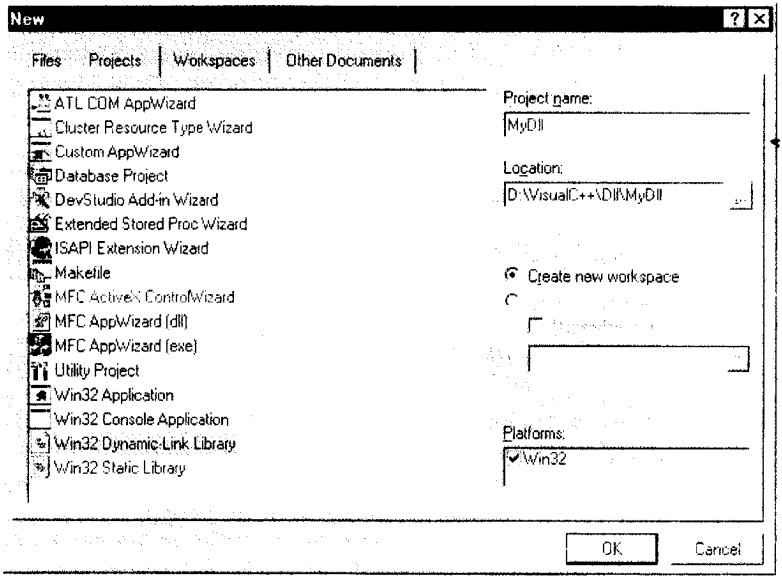


Рисунок 4.2 - Діалогове вікно *New*

1. В головному меню вибрати пункт *File*, задати команду *New...*, а в діалоговому вікні *New*, яке відкриється, вибрати вкладку *Projects*. (Як правило, ця вкладка спочатку завжди відкрита) (рис.4.2).

2. В цьому діалоговому вікні необхідно задати такі параметри:

- в полі *Location* вказати адресу робочої папки проекту, наприклад "*D:\VisualC++\Dll*";
- в полі *Project name* задати ім'я проекту, наприклад *MyDll*;
- в списку типів проекту вибрати тип *Win32 Dynamic-Link Library*;
- натиснути кнопку *Create new workspace*;
- вибрати платформу, для якої створюється проект: *Win32*.
- закрити діалогове вікно *New*.

3. В діалоговому вікні *Win32 Dynamic-Link Library*, що відкриється (рис.4.3), вибрати кнопку-перемикач *Simply DLL* (проста бібліотека DLL) і натиснути кнопку *Finish*.

В результаті буде створено файл, який містить спеціальну функцію *DllMain*(), яка є одночасно і функцією входу, і функцією виходу (будемо її надалі називати функцією входу):

```
BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved )
{
  return TRUE; } .
```

За допомогою функції входу можна виконати ініціалізацію та завершальні роботи після закінчення роботи з бібліотекою DLL.

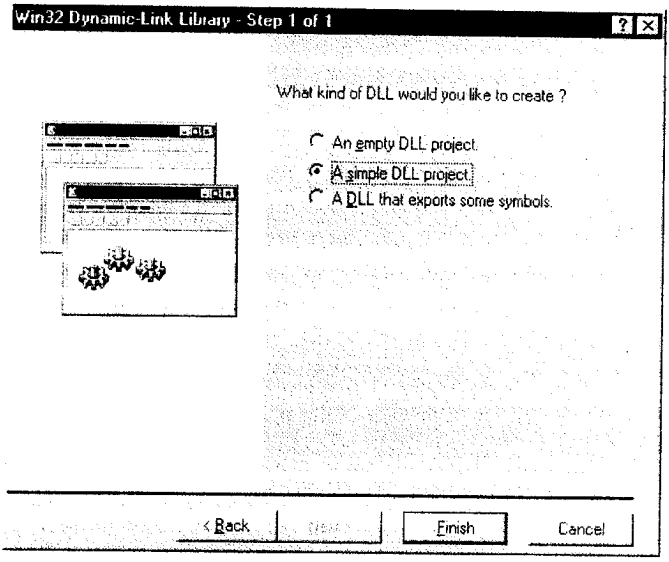


Рисунок 4.3 - Діалогове вікно *Win32 Dynamic-Link Library*

Другий параметр функції входу може мати одне із таких значень:

DLL_PROCESS_ATTACH – означає, що функція викликана процесом як вхідна;

DLL_PROCESS_DETACH – означає, що функція викликана процесом як вихідна;

DLL_THREAD_ATTACH – означає, що функція викликана потоком як вхідна;

DLL_THREAD_DETACH – означає, що функція викликана потоком як вихідна;

Далі у файлі бібліотеки DLL повинна бути описана саме та функція, яка власне і викликається процесами чи потоками. Така функція називається експортованою.

Якщо, наприклад, експортована функція (назвемо її *Calculate*( )) має визначати суму елементів одновимірного масиву, тоді їй необхідно передати два параметри: адресу масиву і розмір масиву. Після виконання функція має повернути обчислену суму. Таким чином у файлі бібліотеки DLL необхідно записати:

```
int Calculate(int *ptr, int n)
{
 int result=0;
 int i,*mas;
   mas=ptr;
for(i=0; i<n; i++)
  result=result+mas[i];
 return result;
}.
```

Досить часто експортовані функції використовують підпрограми мовою Асемблера. В цьому випадку мовою Асемблера, як правило, програмується обчислювальна операція, час виконання якої має бути мінімальним. А операції введення-виведення, які важко реалізуються мовою Асемблера, програмуються мовою високого рівня, вже за межами експортованої функції. Варто відмітити, що такий варіант використання Асемблера з практичної точки зору є найкращим.

Створимо експортовану функцію, що визначає суму елементів одновимірного масиву, але вже мовою Асемблера.

```
int Calculate(int *ptr, int n)
{
int result;
 int address = (int)ptr;
 int four = 4;
      _asm{
            mov ebx,address
            mov   ecx,n
            mov   eax,0
      M1:  add eax,[ebx]
            add ebx,four
            loop   M1
            mov result,eax
           }
      return result;
}.
```

Ім'я експортованої функції потрібно вказати в окремому файлі визначень (файл типу *.def). Створюється файл визначень таким чином.

1. В головному меню вибрати пункт *File*, задати команду *New...*, а в діалоговому вікні *New*, яке відкриється, вибрати закладку *Files*.
2. В цьому діалоговому вікні необхідно вказати такі параметри:

- в полі *Location* вказати адресу файлу;
- в полі *Name* вказати ім'я файлу, наприклад *MyDll.def*;

- в списку типів файлів вибрати тип *Text File*;
- встановити перемикач в положення *Add to Project*.

3. В текстовому редакторі набрати текст файлу визначень, наприклад:

```
LIBRARY "MyDll"
DESCRIPTION 'This library has one function'
EXPORTS
    Calculate @1 .
```

У файлі визначень вказується ім'я експортованої функції та її номер, по якому функцію теж можна викликати.

Крім використання файлу визначень, існує ще один спосіб експорту функцій із бібліотеки DLL. Для цього потрібно визначити експортовану функцію в бібліотеці DLL як глобальну:

extern "C" _declspec(dllexport);

Для остаточного завершення створення власної бібліотеки DLL необхідно виконати такі кроки.

1. В головному меню вибрати пункт *Build*, задати команду *Set Active Configuration*, а в діалоговому вікні, яке відкриється, вибрати ім`я файлу динамічної бібліотеки: *MyDll.dll*.

2. Задати команду *Rebuild*.

В результаті буде створено папку *Release*, де буде знаходитись готовий до роботи файл динамічної бібліотеки *MyDll.dll*. Далі бажано перемістити отриманий файл у папку *Debug*, де знаходиться основний виконувальний файл проекту - файл типу *EXE*.

4.8.3 Підключення функцій бібліотеки DLL до виконувального файлу

Перед викликом експортованої функції необхідно завантажити бібліотеку DLL або за допомогою API-функції *LoadLibrary*(), або за допомогою методу *AfxLoadLibrary*() для MFC-програм. Після успішного завантаження така функція повертає дескриптори об'єктів DLL, за допомогою яких можуть бути викликані експортовані функції динамічної бібліотеки. Для виклику експортованої функції із бібліотеки DLL використовується API-функція *GetProcAddress*():

GetProcAddress(HMODULE hModule, LPCSTR lpProcName); .

Підключити функцію із бібліотеки DLL до виконувального файлу без використання файлу визначень можна, якщо в основному файлі (типу *.cpp) оголосити цю функцію як глобальну імпортовану:

extern "C" _declspec(dllimport); .

Після закінчення роботи з бібліотекою DLL, її необхідно вивантажити за допомогою API-функції *FreeLibrary*().

Повністю фрагмент програми основного файлу (типу *.cpp) матиме такий вигляд.

```
HINSTANCE hDll;
hDll = ::LoadLibrary("MyDll");
if (hDll == NULL)
 MessageBox("file MyDll.dll was not find...");
else
    {
 typedef int (*FUN)(int *,int);
 FUN func;
     func = (FUN)::GetProcAddress(hDll,"Calculate");
          res = (*func)(matrix[m],n);
          wsprintf(s,"%d",res);
          MessageBox(s,"Result after DLL");
          }
     ::FreeLibrary(hDll); .
```

Післямова

З великим жалем необхідно закінчувати цей навчальний посібник. Хотілось ще дуже багато розповісти про програмування в операційних системах Windows 95/98/ME. Наведені в посібнику окремі фрагменти звичайно не можуть охопити всі сторони програмування мовами С та С++. Окремої розмови варті такі питання, як програмування баз даних, технологія ActiveX, COM-технологія, WinInet-програмування, та багато іншого.

Головна мета, яку ставив перед собою автор – показати нові цікаві застосування мов С та С++ в багатозадачних операційних системах. Обмеженість в часі та в об'ємі книги не дозволили автору повністю досягти цієї мети. Тому автор з подякою сприйме всі зауваження та побажання, які виникли при вивченні цього посібника.

Програмування – це не той предмет, який можна вивчити один раз і назавжди. Вивчати нові програмні технології потрібно постійно. Автор хоче побажати успіхів читачам на цьому нелегкому шляху.

ЛАБОРАТОРНИЙ ПРАКТИКУМ

В даному розділі розглядається тематика лабораторних робіт, відповідно до якої студенти попередньо готуються до виконання робіт, користуючись при цьому лекційним матеріалом, матеріалом практичних занять, літературою та відповідними вказівками, викладеними в комп'ютерних файлах.

Лабораторна робота №1

Розробка мінімальної API-програми

Порядок виконання роботи

1. Створити проект типу *"Win32 Application"*.
2. Набрати текст мінімальної API-програми.
3. Виконати мінімальну API-програму при різних значеннях параметрів класу вікна та функції CreateWindow( ).

Лабораторна робота №2

Векторна графіка в API-програмах

Порядок виконання роботи

1. Для заданого варіанта запрограмувати виведення на екран дисплея контурів стандартних геометричних фігур.
2. Зафарбувати замкнуті фігури заданими кольорами та стилями зафарбування.
3. Запрограмувати виведення на екран дисплея складних геометричних фігур з використанням регіонів.

Лабораторна робота №3

Виведення тексту в API-програмах

Порядок виконання роботи

1. Запрограмувати виведення на екран дисплея одного текстового рядка із заданими параметрами.
2. Запрограмувати виведення на екран дисплея кількох текстових рядків із заданими міжрядковими інтервалами.

Лабораторна робота №4

Розробка меню в API-програмах

Порядок виконання роботи

1. Створити проект з одним файлом (типу *.cpp), який містить мінімальну API-програму.

2. Розробити меню за заданим варіантом, використовуючи візуальний редактор меню.

3. Підключити меню до проекту.

4. Написати функції-обробники для пунктів меню.

Лабораторна робота №5

Програмування діалогу в API-програмах

Порядок виконання роботи

1. Створити проект, який містить API-програму з меню.

2. Створити шаблон діалогового вікна за заданим варіантом, використовуючи візуальний редактор діалогу.

3. Підключити діалогове вікно до проекту.

4. Написати діалогову процедуру для ініціалізації елементів діалогу.

5. Доповнити діалогову процедуру для операцій введення-виведення даних з використанням меню.

Лабораторна робота №6

Керування процесами і потоками в API-програмах

Порядок виконання роботи

1. Підготувати тестовий приклад виконувального файлу (наприклад, TEST.EXE)

4. Створити проект, який містить API-програму з меню.

5. В API-програмі створити новий процес для файлу TEST.EXE.

6. В API-програмі створити два нових потоки, які виконуються паралельно.

7. Забезпечити серіалізацію для створених паралельних потоків.

Лабораторна робота №7

Розробка меню в MFC-програмах

Порядок виконання роботи

1. Створити проект з мінімальною MFC-програмою.

2. Розробити меню за заданим варіантом, використовуючи візуальний редактор меню.

3. Підключити меню до проекту.

4. Написати функції-обробники для пунктів меню.

Лабораторна робота №8

Розробка панелі інструментів та рядка стану в MFC-програмах

Порядок виконання роботи

1. Створити проект з MFC-програмою, яка містить меню.

2. Створити за допомогою вбудованого графічного редактора кнопки панелі інструментів, які дублюють пункти меню.

3. Підключити функції-обробники для кожної кнопки панелі інструментів.

4. Підключити панелі інструментів та рядок стану до проекту.

Лабораторна робота №9

Програмування векторної та растрової графіки в MFC-програмах

Порядок виконання роботи

1. Для заданого варіанта запрограмувати виведення на екран дисплея контурів стандартних геометричних фігур.
2. Зафарбувати замкнені фігури заданими кольорами та стилями зафарбування.
3. Створити бітове зображення за допомогою вбудованого графічного редактора.
4. Підключити до програми готове бітове зображення.

Лабораторні роботи № 10

Програмування діалогу в MFC-програмах

Порядок виконання роботи

1. Створити проект з одним файлом (типу *.cpp), який містить мінімальну MFC-програму.
2. Створити шаблон діалогового вікна за заданим варіантом, використовуючи візуальний редактор діалогу.
3. Підключити діалогове вікно до проекту.
4. Написати діалогову процедуру для ініціалізації елементів діалогу.
5. Доповнити діалогову процедуру для операцій введення-виведення даних.

Лабораторна робота №11

Створення функцій мовою Асемблера у форматі DLL-бібліотеки

Порядок виконання роботи

1. Створити проект типу *"Win32 DLL"*.
2. Написати задану бібліотечну функцію з використанням вбудованого асемблера.
3. Створити окремий файл DLL-бібліотеки.
4. Підключити DLL-бібліотеку до виконувального файлу типу EXE.

МЕТОДИЧНІ ВКАЗІВКИ ДО ВИКОНАННЯ КУРСОВОЇ РОБОТИ

Загальні положення

Організація курсового проектування здійснюється відповідно до "Положення про виконання курсових проектів та робіт у ВДТУ", затвердженого Ученою радою ВДТУ 30.12.98 р. Матеріал цього розділу може бути використаний при виконанні курсової роботи з дисципліни "Системне програмне забезпечення" студентами денної та заочної форм навчання.

Курсова робота виконується відповідно до індивідуальних завдань і є самостійною роботою студента, призначеною для закріплення, розширення, узагальнення і практичного використання знань, умінь і навичок, одержаних під час навчання. В процесі курсового проектування студент повинен: розширити і закріпити знання, набуті при вивченні дисциплін "Програмування", "Системне програмування" і "Системне програмне забезпечення"; навчитися самостійно і творчо застосовувати отримані знання для розв'язання конкретної інженерної задачі; отримати практичні навички в написанні та тестуванні багатофайлових проектів; опанувати сучасні програмні пакети; навчитись користуватись довідниковою літературою.

Відповідно до затвердженого графіка студент зобов'язаний своєчасно подавати керівникові результати роботи над курсовою роботою.

Завдання на курсову роботу видаються студентам викладачем на першому плановому практичному занятті. Варіанти завдань є індивідуальними для кожного студента і не повторюються.

Завдання можуть бути також спеціалізованими. Спеціалізовані курсові роботи незалежно від конкретного завдання також повинні передбачати програмування мовою C++ та мовою Асемблера. Спеціалізоване завдання повинно бути узгодженим з керівником і консультантом курсової роботи і затвердженим на засіданні кафедри.

Типове завдання на курсову роботу

Тема: Розробка об'єктно-орієнтованих програм мовою C++
в середовищі Windows (з використанням пакету Visual C++ 6.0)

1. Розробити в складі проекту головний файл (файл типу *.cpp), який повинен:

- видавати інформацію про розробника проекту;
- вводити в діалоговому режимі вхідні дані з клавіатури;
- виводити результати роботи програми в числовому та графічному вигляді у різних діалогових вікнах;
- виконувати задані стандартні операції з файлами: відкриття вибраного файлу, збереження вибраного файлу тощо.

2. Розробити в складі проекту ресурсний файл (файл типу *.rc), який повинен включати такі види ресурсів: меню, панель інструментів, рядок стану, діалогові вікна.

3. Розробити в складі проекту динамічно підключувану бібліотеку (файл типу *.dll), яка має містити експортовану функцію мовою Асемблера.

Завдання на курсову роботу мають індивідуальні варіанти для кожного студента:

- варіанти форм меню;
- варіанти вигляду діалогових вікон;
- варіанти завдань для реалізації мовою Асемблера;
- варіанти стандартних операцій з файлами.

Зміст пояснювальної записки до курсової роботи

При оформленні тексту пояснювальної записки до курсової роботи слід користуватися рекомендаціями стандарту ДСТУ 3008-95, де встановлені вимоги до оформлення звітної наукової документації.

Пояснювальна записка повинна включати: титульний лист; завдання на курсову роботу з конкретним варіантом, анотацію, зміст; вступ; розділи курсової роботи, висновки, перелік використаної літератури, додатки (тексти програм).

Кожен розділ повинен містити короткий теоретичний вступ і детальний опис програмної реалізації одного етапу курсової роботи: створення меню, створення панелі інструментів і рядка стану, розробка діалогових вікон, програмування графіки, робота з файлами, створення DLL-бібліотеки і т. д.

Текст пояснювальної записки має бути викладений в лаконічному обгрунтувальному стилі.

Пояснювальна записка виконується на листах формату А4 із стандартною рамкою. Орієнтовний обсяг пояснювальної записки 15-20 аркушів друкованого тексту (без додатків).

Захист курсових робіт

Перед захистом курсової роботи пояснювальна записка подається на перевірку. Якщо пояснювальна записка відповідає всім вимогам діючих стандартів і завдання відповідає заданому варіанту, тоді студент допускається до захисту курсової роботи. Пояснювальні записки, виконані шляхом копіювання інших пояснювальних записок, до розгляду не приймаються.

Захист робіт проводиться на комп'ютері перед комісією з двох викладачів в присутності студентів групи. В результаті захисту курсова робота оцінюється дванадцятибальною оцінкою і відповідною їй модульною оцінкою за модульно-рейтинговою системою в залежності від якості виконання роботи, якості її оформлення, повноти досліджень та рівня відповідей на запитання при захисті роботи.

Література

1. Г.С.Цейтин. На пути к сборочному программированию// Программирование. - 1990. №1. С.78-92.
2. Гради Буч. Объектно-ориентированное программирование с примерами применения: Пер. с англ.-М.:Конкорд,1992.-519с.
3. М.Уэйт, С.Прата, Д.Мартин. Язык Си. Руководство для начинающих: Пер. с англ.-М.: Мир,1988.-512с.
4. Гладков С.А., Фролов Г.В. Программирование в Microsoft Windows: В 2-х ч.-М.:"Диалог-МИФИ",1992.-320с.,288с.
5. Г.Шилдт. Программирование на С и С++ для Windows 95:Пер. с англ.- К.:Торгово-издательское бюро HBV,1996.-400с.
6. Г.Шилдт. Самоучитель С++: Пер. с англ.-СПб.:HBV-Санкт-Петербург, 1997.-512с.
7. М.Луис. Visual C++6.-М.:Лаборатория Базовых Знаний, 1999.-720с.-(справочник).
8. С.Холзнер. Visual C++6.: Учебный курс-СПб.:Питер,2001.-576с.
9. Ю.Тихомиров. Visual C++6.-СПб.:BHV-Санкт-Петербур,1999,496с.
10. А.Мешков, Ю.Тихомиров. Visual C++ и MFC. Программирование для Windows NT и Windows 95: В 3-х томах.-СПб.:BHV-Санкт-Петербург, 1997.-464с.
11. К.Паппас, У.Мюррей. Visual C++6: для пользователя: Пер. с англ.-К.: Издательская группа BHV, 2000.-336с.
12. М.Янг. Векторная графика на языке С++/ Пер. с англ.-М.: Восточная Книжная Компания, 1997.-368с.
13. Гринзоу. Философия программирования для Windows 95/NT: Пер. с англ.- СПб: Символ ПМОС, 1997.-640с.

Навчальне видання

Семеренко В. П.

Програмування мовами С та С++ в середовищі Windows

Навчальний посібник

Оригінал-макет підготовлено автором

Редактор С. А. Малішевська

Підписано до друку 10.09.2002р.
Формат 29,7х42 1/4 Гарнітура Times New Roman
Друк різографічний Ум. друк. арк. 5,43
Тираж 75 прим.
Зам. № 2002-157

Віддруковано в комп'ютерному інформаційно-видавничому центрі Вінницького державного технічного університету
21021, м. Вінниця, Хмельницьке шосе, 95, ВДТУ, ГНК, 9-й поверх
Тел. **(0432) 44-01-59**