Учреждение образования
«Брестский государственный технический университет»

Факультет электронно-информационных систем
Кафедра «Интеллектуальные информационные технологии»

СОГЛАСОВАНО
Заведующий кафедрой
_____ В. А. Головко
«25» 06 2025 г.

СОГЛАСОВАНО
Декан факультета
_____ В.С. Разумейчик
«25» 06 2025 г.

ЭЛЕКТРОННЫЙ УЧЕБНО-МЕТОДИЧЕСКИЙ КОМПЛЕКС ПО УЧЕБНОЙ ДИСЦИПЛИНЕ

«ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ИНТЕЛЛЕКТУАЛЬНЫХ СИСТЕМ»

для специальности:

6-05-0611-03 Искусственный интеллект

Составители: Муравьев Геннадий Леонидович, к.т.н., доцент
Савицкий Юрий Викторович, к.т.н., доцент

Рассмотрено и утверждено на заседании Научно-методического Совета университета от 26 июня 2025 г., протокол № 4.

Рег. № УМК 24/25-318

Пояснительная записка

Актуальность изучения дисциплины

Дисциплина «Проектирование программного обеспечения интеллектуальных систем» является одной из основных в процессе подготовки студентов специальности 6-05-0611-03 Искусственный интеллект в области технологий разработки программ. Носит системный характер, формируя основу для освоения ряда последующих курсов, обеспечивает выработку базовых знаний и умений по основам проектирования программ, имеет выраженную практическую направленность. Относится к дисциплинам компонента учреждения образования, читается в двух семестрах.

Цель дисциплины

Цель преподавания учебной дисциплины: изучение принципов, подходов, методов, средств, технологий проектирования и реализации проектных решений, развития программ, программных систем.

Задачи изучения дисциплины

- приобретение знаний о языках программирования разного уровня, основах парадигм программирования, средствах разработки интеллектуальных систем;
- изучение принципов построения программных систем на основе языков программирования разного уровня;
- формирование практических умений и навыков по использованию методов и технологий проектирования и программирования систем с применением современных ИТ.

Электронный учебно-методический комплекс (ЭУМК) объединяет структурные элементы научно-методического обеспечения процессов обучения, представляет систему материалов теоретического и практического характера, необходимую для организации работы студентов специальности Искусственный интеллект (дневная форма получения образования) по изучению указанной дисциплины. Предлагаемый ЭУМК предназначен для изучения второй части дисциплины - раздел Конструирование программ (подходы, средства).

ЭУМК разработан на основании “Положения об учебно-методическом комплексе на уровне высшего образования”, утвержденного Постановлением Министерства образования Республики Беларусь от 8 ноября 2022 г. № 427, и в соответствии с требованиями учебной программы Учреждения образования «Брестский государственный технический университет», утвержденной 28.06.2024 г., регистрационный № УД-24-1-303/уч. по дисциплине «Проектирование программного обеспечения интеллектуальных систем».

Цели ЭУМК: - повышение эффективности образования за счет качественного методического сопровождения процессов обучения; - обеспечения самостоятельной работы студентов; - внедренияе в учебный процесс информационных технологий.

Актуальность создания ЭУМК по курсу «Проектирование программного обеспечения интеллектуальных систем» обусловлена:

- важностью дисциплины для подготовки специалистов в области разработки ПО, использования информационных технологий;
- необходимостью комплексного, системного и целостного обеспечения учебного процесса как актуальными теоретическими знаниями так и руководством по получению практичеких умений и навыков;

- тенденцией поддержки единого стандарта обучения.

Содержание и объем ЭУМК соответствуют образовательному стандарту высшего образования специальности 6-05-0611-03 Искусственный интеллект, а также учебно-программной документации образовательных программ высшего образования. Материал представлен на требуемом методическом уровне и адаптирован к современным образовательным технологиям.

Структура ЭУМК по дисциплине «Проектирование программного обеспечения интеллектуальных систем»

Теоретический раздел содержит материалы для теоретического изучения учебной дисциплины и представлен конспектом лекций.

Практический раздел содержит материалы для проведения лабораторных учебных занятий в виде примерного перечня тем лабораторных занятий и методических указаний для их выполнения.

Раздел контроля знаний содержит материалы для итоговой аттестации (экзаменационные вопросы), позволяющие определить соответствие результатов учебной деятельности обучающихся требованиям образовательных стандартов высшего образования и учебно-программной документации образовательных программ высшего образования.

Вспомогательный раздел включает учебную программу учреждения высшего образования по учебной дисциплине «Проектирование программного обеспечения интеллектуальных систем».

Рекомендации по организации работы с ЭУМК:

- лекции проводятся с использованием представленных в ЭУМК учебных теоретических материалов, графического материала (схем, рисунков);

- лабораторные занятия проводятся в условиях учебной аудитории, оборудованной средствами вычислительной техники, с использованием представленных в ЭУМК методических указаний к их выполнению. Студенты могут использовать конспект лекций ЭУМК и указанные методические указания при подготовке и в ходе лабораторных занятий;

- при подготовке к экзамену студенты могут использовать конспект лекций ЭУМК, экзаменационные материалы, которые приведены в разделе контроля знаний ЭУМК.

ПЕРЕЧЕНЬ МАТЕРИАЛОВ В КОМПЛЕКСЕ

СОДЕРЖАНИЕ

1 ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ

Конспект лекций

Лекция № 1 Средства разработки оконных приложений C++

1 Особенности программирования в ОС Windows

При разработке программного обеспечения, ориентированного на исполнение с участием операционной системы Windows, говорят о разработке Windows-приложений или просто приложений (либо, на момент разработки, т.е. до получения готового к исполнению продукта - говорят о разработке проекта приложения – project). Здесь существует два основных подхода (технологических направления) к разработке приложений.

Первый подход - процедурная разработка (процедурное проектирование, программирование, тестирование), базирующаяся на алгоритмической декомпозиции предметной области (автоматизируемых задач) и принципах структурной разработки программ. При разработке Windows-приложений этот подход называют также стилем низкоуровневого программирования, так как он связан с непосредственным использованием функций операционной системы Windows - Win32 API. Результатом применения этого стиля в среде Visual Studio являются Windows-приложения. Это API-приложения как “консольного” так и “оконного” типов.

Второй подход – объектно-ориентированная разработка (проектирование, программирование, тестирование), базирующаяся на объектной декомпозиции предметной области и принципах объектно-ориентированной модели ПО. При разработке Windows-приложений этот подход называют также стилем высокоуровневого программирования, так как он не ориентирован на непосредственное использование функций операционной системы Windows (хотя и допускает их прямое использование), а базируется на использовании библиотек готовых классов (где и эти функции инкапсулированы в соответствующих классах) и пользовательские классы. Это, например, следующие библиотеки: - MFC (Microsoft Foundation Class Library), используемая для создания сложных Windows-приложений с богатым графическим интерфейсом; - ATL (Microsoft Active Template Library) библиотека шаблонов ActiveX для создания COM-объектов и элементов управления ActiveX; - STL (Standard Template Library) стандартная библиотека шаблонов. Результатом применения этого стиля в среде Visual Studio являются Windows-приложения. Это оконные MFC-приложения как “статичного” так и “динамического” типов.

ОС Windows как виртуальная машина. Операционная система Windows базируется на понятии виртуальная машина. Она воплощает принципы обеспечения независимости пользователя от особенностей конкретных аппаратных средств системы, а также деталей реализации программных компонентов и функций предоставляемого ею программного обеспечения. Это принципы сокрытия от пользователя деталей управления предоставляемыми ресурсами, принципы стандартизации средств управления. Указанная идеология находит отражение в интерфейсе управления ОС - GDI (Graphics Device Interface). Он обеспечивает программиста средствами выстраивания стандартных (оконных) интерфейсов и средствами их реализации (функциями API взамен функций DOS).

Windows отличается так же обеспечением многозадачности. Под его управлением одновременно может выполняться много приложений. Более того, в рамках приложений могут быть определены параллельно выполняемые и при необходимости синхронизируемые участки (потоки). Указанное поддерживается аппаратом DLL-функций (реентерабельных модулей), автоматическим управлением памятью системы в защищенном

режиме (на базе виртуальной – линейной, плоской модели памяти в 4 Гб с использованием механизмов страничного скроллинга), стратегией вытесняющей многозадачности с разнообразными дисциплинами обслуживания решаемых задач, включая механизм разделения времени типа RR.

При этом, все выполняемые оконные приложения общаются с другими ресурсами (информационными, аппаратными, программными), с внешним миром (пользователями, ОС, аппаратурой, другими приложениями и т.д.) посредством единого механизма пересылки и обработки сообщений (о происходящих событиях) через одно центральное звено – ОС Windows. Сообщение, полученное приложением, может игнорироваться или обрабатываться в зависимости от задач, решаемых программистом. После завершения обработки текущего сообщения приложение бездействует до прихода следующего сообщения. Обработка сообщения состоит в отработке определенных программистом (пользователем) действий, описанных в приложении.

2 Особенности Windows-приложений на языках C, C++

При программировании приложений традиционно существуют отличия и особенности использования средств языка С и языка С++.

Есть особенности в написании Windows-приложений обусловленные спецификой 16 или 32 разрядной реализаций ОС Windows. Это: - несовпадение разрядностей одних и тех же типов (например, тип int может использовать 16 или 32 бита); - несовпадение типов указателей (в DOS и 16 разрядной реализаций ОС Windows используются как 16 битовые указатели типа near, так и 32 битовые указатели типа far и huge, а в 32 разрядной реализаций ОС Windows все указатели являются ближними - near и занимают 32 бита).

Есть особенности, определяемые самой ОС Windows, связанные с уже упоминавшейся особой структурой приложений, особыми типами функций, обработкой сообщений и организацией взаимодействия через сообщения. Это использование большого числа новых заголовочных файлов, подключаемых, в частности, через хедер windows.h автоматически. Кроме этого существуют и стилевые особенности:

- использование переопределенных, дополнительных имен для обозначения типов данных языка С, С++ (наряду со стандартными именами); - использование новых, специфических типов (имен типов) для новых объектов-данных и указателей (дескрипторов, типов результатов, специфических типов указателей на наиболее распространенные типы данных общего назначения и т.п.), определенных на базе стандартных типов данных С++ ; - широкое применение венгерской нотации - правил записи имен переменных, объектов в соответствии с принципами структурного программирования; - использование специфичных структур данных как для организации функционирования приложений так и для организации пользовательских функций.

Заголовочные файлы Windows. В системе Visual Studio используется много новых заголовочных файлов (хедеров), подключаемых к приложению. Как правило, если приложение создается программой-мастером на базе специального шаблона, называемого заготовкой, каркасом приложения, то большинство необходимых приложению заголовочных файлов включаются в него мастером автоматически в том числе, например, и через хедер windows.h для оконных приложений. Состав включения зависит от выбранного в системе программирования типа создаваемого приложения и его атрибутов. Заголовочные файлы (несколько сотен) находятся в папке Include системы.

Общая характеристика наиболее часто используемых хедеров дана ниже. Так заголовочный файл windows.h регулирует в зависимости от версии ОС и других атрибутов подключение дополнительных заголовочных файлов, например,

```
#include <windef.h>
#include <winbase.h>
#include <wingdi.h>
#include <winuser.h>
…
#include <wincon.h>
#include <winver.h>
…
```

```
#ifndef  NOGDI
#include <commdlg.h>
#ifndef _MAC
 #include <winspool.h>
#ifdef INC_OLE1
#include <ole.h>
 #else
#include <ole2.h>
 #endif    ,
```

которые в свою очередь могут подключить следующие заголовочные файлы.

Заголовочный файл windowsx.h осуществляет подключение определений макрофункций API, обработчиков оконных сообщений и функций элементов управления.

Заголовочный файл windef.h определяет новые типы данных для специфичных объектов Windows, например, DWORD, BOOL, BYTE, WORD, PFLOAT и другие.

В заголовочном файле winbase.h определены структуры, прототипы базовых API функций, необходимых для управления ресурсами 32 разрядной реализаций ОС Windows. Например, структуры SYSTEMTIME, PROCESS_INFORMATION, прототипы функций WinMain, CreateThread, WriteFile, ReadFile, GetSystemTime и т.д.

В заголовочном файле wingdi.h декларированы прототипы GDI функций, константы и макросы, необходимые для работы с экраном (операций ввода-вывода), например, структуры BITMAP, TEXTMETRICA, функции Arc, CreatePalette, CreatePen и другие.

В заголовочном файле winuser.h определены прототипы функций, константы и макросы, используемые программистом в пользовательских приложениях, например, функции WNDPROC, DLGPROC и другие.

Кроме этого, в папке Include представлены специфические хедеры типа cmath, cstdio, cctype, climits, cstring и другие, позволяющие подключать в новом стиле стандартные функции из старых библиотек. Например, для подключения математических функций, описанных хедером math.h, используется директива #include <cmath>. Здесь хедер cmath включает фрагмент для подключения хедера math.h

```
 #ifdef     _STD_USING
    #undef   _STD_USING
    #include <math.h>
    #define _STD_USING
#else
   #include <math.h>
#endif    .
```

3 Типы данных Windows

Дополнительные имена для обозначения типов данных языка C, C++ (наряду с их стандартными именами), введенные путем переопределения командой typedef стандартных названий, представлены ниже:

```
typedef unsigned long   DWORD;      typedef CONST void far     *LPCVOID;
typedef int             BOOL;       typedef int                 INT;
typedef unsigned char   BYTE;       typedef unsigned int        UINT;
typedef unsigned short  WORD;       typedef unsigned int       *PUINT;
```

```
typedef float          FLOAT;      typedef unsigned long    ULONG;
typedef FLOAT          *PFLOAT;    typedef ULONG            *PULONG;
typedef BOOL near      *PBOOL;     typedef unsigned short   USHORT;
typedef BOOL far       *LPBOOL;    typedef USHORT           *PUSHORT;
typedef BYTE near      *PBYTE;     typedef unsigned char    UCHAR;
typedef BYTE far       *LPBYTE;    typedef UCHAR            *PUCHAR;
typedef int near       *PINT;      typedef UINT             WPARAM;
typedef int far        *LPINT;     typedef LONG             LPARAM;
typedef WORD near      *PWORD;     typedef LONG             LRESULT;
typedef WORD far       *LPWORD;    typedef WORD             ATOM;
typedef long far       *LPLONG;    typedef int              HFILE;
typedef DWORD near     *PDWORD;    typedef void far         *LPVOID;
typedef DWORD far      *LPDWORD .
```

Типы данных общего назначения, определенные на базе стандартных типов данных C++ командой typedef (заголовочный файл windef.h), представлены ниже (таблица 1).

Таблица 1 - Типы данных общего назначения

Тип данного	**Тип-аналог C++**	**Раз-мер, бит**	**Описание значений**	**Диапазон**
BOOL	—	32	логическое значение	TRUE (1), FALSE (0)
BOOLEAN	—	32	логическое значение	TRUE(1), FALSE(0)
BYTE	unsigned char	8	байт без знака для хранения числа или кода символа	0...255
CCHAR	char	8	символ Windows	-128...+127
CHAR	char	8	символ Windows	-128...+127
CONST	const	-	константа	-
DWORD	unsigned long	32	двойное слово без знака	0... 42944967295
DWORD-LONG	double	64	число с плавающей точкой со знаком	1.7E-308... 1.7E+308
FLOAT	float	32	число с плавающей точкой со знаком	3.4E-38... 3.4E+38
INT	int, long	32	целое число со знаком	-2147483648... +2147483647
LONG	long, int	32	целое число со знаком	-2147483648... +2147483647
LONGLONG	double	64	число с плавающей точкой со знаком	1.7E-308... 1.7E+308
SHORT	short	16	короткое целое число со знаком	-32768... +32767
TBYTE	unsigned char	8	байт без знака для хранения числа или кода символа	0...255
TCHAR	char	8	символ Windows или Unicode	-128...+127
UCHAR	unsigned char	8	символ Windows без знака	0...255
UINT	unsigned int	32	целое число без знака	0... 4294967295

ULONG	unsigned long	32	целое число без знака	0... 4294967295
USHORT	unsigned short	16	короткое целое число без знака	0...65535
VOID	void	-	любой тип	-
WCHAR	wchar_t	16	символ Unicode	0...65535
WORD	—	16	короткое целое число без знака	0...65535

Специфические типы указателей на наиболее распространенные типы данных общего назначения, определенные на базе стандартных типов данных C++ командой typedef (заголовочный файл windef.h), представлены ниже (таблица 2).

Таблица 2 - Типы указателей на типы данных общего назначения

Обозначение указателей	**Тип адресуемого данного**	**Примечание**
LPBOOL, PBOOL	BOOL	
LPBYTE, PBYTE	BYTE	
LPCCH, PCCH	CONST CHAR	константный символ
LPCH, PCH	CHAR	символ
LPCSTR, PCSTR	CONST CHAR	константная строка с завершающим нулем
LPCTSTR	CONST TCHAR	константная строка символов Windows или Unicode с завершающим нулем
LPCWCH, PCWCH	CONST WCHAR	константный символ Unicode
LPCWSTR, PCWSTR	CONST WCHAR	константная строка Unicode с завершающим нулем
LPDWORD, PDWORD	DWORD	
LPINT, PINT	INT	
LPLONG, PLONG	LONG	
LPSTR, PSTR	CHAR	строка символов с завершающим нулем
LPTCH, PTCH	TCHAR	символ Windows или Unicode
LPTSTR, PTSTR	TCHAR	строка символов Windows или Unicode с завершающим нулем
LPVOID, PVOID	VOID	
LPWCH, PWCH	WCHAR	символ Unicode
LPWORD, PWORD	WORD	
LPWSTR, PWSTR	WCHAR	строка Unicode с завершающим нулем
NPSTR	CHAR	строка символов с завершающим нулем
PBOOLEAN	BOOL	
PCHAR	CHAR	символ Windows
PFLOAT	FLOAT	
PSHORT	SHORT	
PSZ	CHAR	строка символов с завершающим нулем
PTBYTE	TBYTE	символ Windows или Unicode

PTCHAR	TCHAR	символ Windows или Unicode
PUCHAR	UCHAR	символ Windows без знака
PUINT	UINT	
PULONG	ULONG	
PUSHORT	USHORT	
PWCHAR	WCHAR	символ Unicode

Примеры новых типов для описания специфичных объектов Windows, определенных на базе стандартных типов данных C++ (см. заголовочный файл windef.h), представлены ниже. Например, команда typedef LONG LRESULT определяет тип LRESULT, используемый как тип результата работы функции-обработчика сообщений окна; typedef int HFILE определяет тип HFILE, используемый в качестве дескриптора файла и т.п.

Венгерская нотация. Префиксы идентификаторов. Для повышения читабельности текстов приложений за счет осмысленности используемых в них идентификаторов (имен) в ОС Windows и приложениях Windows применяется венгерская нотация. Это правила записи имен переменных, объектов в соответствии с принципами структурного программирования.

Таблица 3 - Типовые префиксы венгерской нотации

Префикс	**Полный префикс**	**Смысловое значение префикса**
b	Bool	логическая переменная
c	Character	символ, 1 байт
dw	DoubleWord	двойное слово без знака, 32 бита (целое число)
f / fn	Function	функция
pfn	PointerFunction	указатель на функцию
lpfn	LongPointerFunction	длинный указатель на функцию
h	HANDLE	дескриптор объекта
hDC	HANDLE	дескриптор контекста устройства
id		значение ID-идентификатора
l	LONG	длинное целое со знаком, 32 бита
lp	LongPointer	дальний указатель, 32 бита
lpsz	LongPointerString-Zero	дальний указатель на строку, заканчивающуюся нуль-символом, 32 бита
n	iNt	короткое целое число со знаком
p / np	Pointer	ближний указатель, 32 бита
pt	PoinT	x и y координаты точки, упакованные в 64 бита
s	String	строка
sz	StringZero	символьная строка, заканчивающаяся нуль-символом
pst	PointerStruct	указатель на структуру
psz	PointerStringZero	указатель на строку, заканчивающуюся нуль-символом
u	Uint	беззнаковый символ, целое без знака
w	WORD	беззнаковое значение, 16 бит
by	BYTE	беззнаковый символ
i	Integer	целое число, 32 бита
pv	PointerVoid	указатель на тип void
v	Void	тип void
W	Wide	символ UNICODE, 16-бит

Соответственно при описании переменных в C++ при создании Windows-приложений используются специальные правила формирования имен на базе префиксов (таблица 3). Здесь

префиксное_ИМЯ_ПЕРЕМЕННОЙ ::= <префикс> < основное_ИМЯ_ПЕРЕМЕННОЙ >,

где префиксное_ИМЯ_ПЕРЕМЕННОЙ формируется по правилам венгерской нотации и явно определяет его типовую принадлежность; <префикс>, описываемый строчными буквами, задает тип переменной в виде одного из типовых сокращений; < основное_ИМЯ_ПЕРЕМЕННОЙ > представляет собой составное мнемоническое имя переменной, записанное строчными символами, без подчеркиваний, каждая часть имени пишется с заглавной буквы.

Ниже приведены примеры описания имен, используемых в различных приложениях:

int nCmdShow;	LPSTR lpszCmdLine;	HINSTANCE hInst;
int cbClsExtra;	LPSTR szCmdLine;	HANDLE hInst;
UINT nMessage;	LPCSTR lpszClassName;	HWND hWnd;
DWORD dwStyle;	LPCSTR szProgName;	HICON hIcon;
	LPCSTR lpszMenuName;	HCURSOR hCursor;
MSG lpMsg;		HMENU hMenu;
WPARAM wParam;	WNDPROC lpfnWndProc;	
LPARAM lParam; .		

Лекция № 2 Приложения, управляемые сообщениями

1 Особенности оконных приложений Windows

Такие приложения предлагают пользователю привычный для ОС Windows интерфейс на базе системы, иерархии окон. Это окна различных типов (“классические” всплывающие окна с клиентской областью, пользовательские и стандартные диалоговые окна, включая специальные окна типа Save As и т.п.) и расположенные в них элементы управления (меню, кнопки, списки и т.п.). Одно из окон является “главным”, поскольку запускается первым. Теперь происходящие сообщения относятся именно к нему. В частном случае приложение может не иметь окон. Типичное оконное Windows-приложение в качестве главного окна выводит “классическое” всплывающее окно. При необходимости в роли главного окна может использоваться диалоговое окно.

Соответственно типичное оконное Windows-приложение, написанное, например, на языке C++, отличается специфической структурой, предусматривающей использование специальной глобальной функции WinMain (аналог функции main языка C) и, как минимум, одной функции типа обработчик сообщений окна. Это, как правило, обработчик сообщений главного окна. Функция WinMain является точкой первого входа в приложение после его запуска пользователем. Она производит интерфейсные настройки и затем переходит в режим ожидания и диспетчирования потока сообщений. А функция обработчик (главного окна), относящаяся к особому типу функций обратного вызова, запускается со стороны ОС посылкой сообщения, обрабатывает его, ждет следующих сообщений.

Эта функция может включать вызовы других функций на языке С, в том числе вызовы библиотечных функций С, С++, вызовы функций Windows, пользовательских функций.

Ресурсы Windows-приложений. В операционной системе Windows специальные графические объекты, имеющие облик и используемые для организации и поддержки графических интерфейсов пользователя, называются ресурсами. Как правило, это окна различных типов. Это диалоговые окна (типовые и пользовательские), служащие подложкой, где могут располагаться различные элементы управления и сами элементы управления. Стандартные диалоговые окна Windows (см. commdlg.h) – часто используемые окна типа Сохранить как, Открыть, Печать, Параметры страницы и др. Основные ЭУ – кнопки, переключатели, списки, комбинированные, графические списки, тексты, групповые рамки, линии прокрутки, линейки с ползунками, окна редактирования, альтернативные кнопки, флажки, индикаторы и т.д.

С пользовательской точки зрения ресурсы - это изображения графических объектов, формирующих визуальное представление интерфейса приложения. Предназначены для организации канала общения пользователь-приложение, используемого как для ввода информации (данных и управляющих воздействий пользователя) так и вывода информации. Построены в соответствии со стандартом GDI.

С информационной точки зрения это описания графических объектов, достаточные для генерации соответствующего ресурса. Это текстовые описания (например, описания меню, диалоговых окон), выполненные в терминах специальных декларативных команд. Это описания, представленные в графическом формате (например, пиктограммы, битовые образы). Это описания, представленные настроечными данными в специальных структурах данных (например, окна), достаточными для генерации соответствующего ресурса.

С компонентной (модульной) точки зрения - это файлы описаний ресурсов, не разделяемые с другими приложениями. В текстовом, не откомпилированном виде описания ресурсов хранятся в файле с расширением rc, а в откомпилированном виде в файле с расширением res.

С точки зрения реализации - это автоматически генерируемое программное обеспечение, обеспечивающее визуализацию ресурсов и работу с ними.

2 Система VISUAL STUDIO. Каркасное программирование

Visual Studio — это система программирования, комплект (suite) средств разработки, включающий язык Visual C++, комплекс программ, объединенных в интегрированную среду разработки (Integrated Development Environment), и библиотеки функций. Visual Studio ориентирован на создание как автономных (работающих на отдельной ПЭВМ) так и сетевых приложений под Windows. Это оконные Windows-приложения (в частном случае консольные приложения), создаваемые с использованием как технологии процедурной разработки программ (стиль низкоуровневого программирования) так и технологии объектно-ориентированной разработки программ (стиль высокоуровневого программирования). Соответственно Visual Studio содержит в своем составе два набора программных средств для поддержки указанных технологий, включающих, помимо типового набора средств системы программирования, также готовые каркасы типовых приложений и мастера для их создания.

Каркас приложения - это готовая заготовка для каждого типового Windows-приложения (например, каркас консольного приложения и т.п.). Выбор, предварительная настройка конкретного типа каркаса и автоматическая генерация его текста производится с помощью мастеров построения каркасов. Результат – программный текст. Само же

программирование в Visual Studio называется каркасным, поскольку базируется на использовании каркасов типовых Windows-приложений с их последующим до программированием.

Результатом применения стиля низкоуровневого программирования в системе Visual Studio являются API-приложения как “консольного” (тип проекта в системе Visual Studio – “Win32 Console Application”) так и “оконного” (тип проекта в системе Visual Studio – “Win32 Application”), а также смешанного типов. При создании таких приложений с помощью средств системы Visual Studio можно использовать следующие типовые каркасы:

1) для консольного API-приложения (тип проекта в системе Visual Studio – “Win32 Console Application”) каркасы типа пустой - Empty (“An empty project”), простой - Simple (“A simple application”), простой с выводом приветствия - Hello (“A “Hello, World” application”);

2) для смешанного консольного API-приложения (тип проекта в системе Visual Studio – “Win32 Console Application”) каркас типа API-MFC (“An application that supports MFC”);

3) для оконного API-приложения (тип проекта в системе Visual Studio – “Win32 Application”) каркасы типа пустой - Empty (“An empty project”), простой - Simple (“A simple Win32 application”), Hello (“A typical “Hello, World” application”).

Результатом применения стиля высокоуровневого программирования в системе Visual Studio являются “статические” (тип проекта в системе Visual Studio – “MFC AppWizard(exe)”) и “динамические” (тип проекта в системе Visual Studio – “MFC AppWizard(dll)”) оконные MFC-приложения. При создании таких приложений с помощью системы Visual Studio можно использовать следующие типовые каркасы:

1) для статического MFC-приложения (тип проекта в системе Visual Studio – “MFC AppWizard(exe)”) каркасы с однодокументным интерфейсом (“Single document”), с многодокументным интерфейсом (“Multiple documents”), с интерфейсом в виде диалогового окна (“Dialog based”);

2) для динамического MFC-приложения (тип проекта в системе Visual Studio – “MFC AppWizard(dll)”) доступны каркасы “Regular DLL with MFC statically linked”, “Regular DLL using shared MFC DLL), “MFC extension DLL (using shared MFC DLL)).

Лекция № 3 Порядок функционирования приложения

1 Общая структура оконных приложений

С точки зрения пользователя (на уровне применения) приложение может быть описано законом функционирования с помощью поведенческих моделей. Например, в терминах перечня решаемых задач, соответствующей входной и выходной информации и правил получения выходной информации из входной без раскрытия способов реализации задач (закона функционирования приложения) как на рисунках 1, 2. Каждая из решаемых задач может быть детализирована в виде сценария использования приложения в терминах поддерживаемого им графического интерфейса пользователя.

На рисунках здесь и далее сплошными линиями показаны направления передачи управления от модуля к модулю (например, от ОС к приложению), пунктирным линиям соответствуют направления передачи сообщений (например, от ОС к приложению и обратно). Ломаные стрелки отображают действия, события, инициирующие сообщения (, например, действия пользователя - нажатие клавиши, перемещение “мыши” и т.п.), полые стрелки показывают направления движения информации, данных (например, от пользователя к приложению и в обратно).

В структуре приложения (рисунок 3) можно выделить часть, видимую пользователем – интерфейс приложения, и сами программы, поддерживающие интерфейс и выполняющие обработку данных в соответствии с принятыми сообщениями, - реализация приложения. Интерфейс оконного приложения (рисунок 4) в общем случае может быть представлен, как правило, системой окон различных классов и типов с расположенными на их поверхности элементами управления. Пользователь может воздействовать на элементы управления (например, нажатием кнопки, выбором пункта меню и т.д.) и тем самым посылать сообщения. При загрузке приложения, как правило, выводится начальное (главное) окно приложения и начинает работать связанная с ним программа обработчик сообщений, адресуемых этому окну. При этом возможна передача управления (активности) одному из других окон интерфейса (обработчику его сообщений). В текущем времени только одно из окон приложения является активным, а следовательно, все полученные в этот момент сообщения адресуются именно этому окну (точнее на обработку программе обработчику сообщений активного окна). Сообщения приложению направляются со стороны ОС.

Более детально Windows-приложение может быть специфицировано в терминах обрабатываемых сообщений. Здесь поведенческое описание может представлять спецификацию сообщений, направляемых приложению, и спецификацию реакций приложения на полученные сообщения (при этом приложение по-прежнему рассматривается как “черный” ящик).

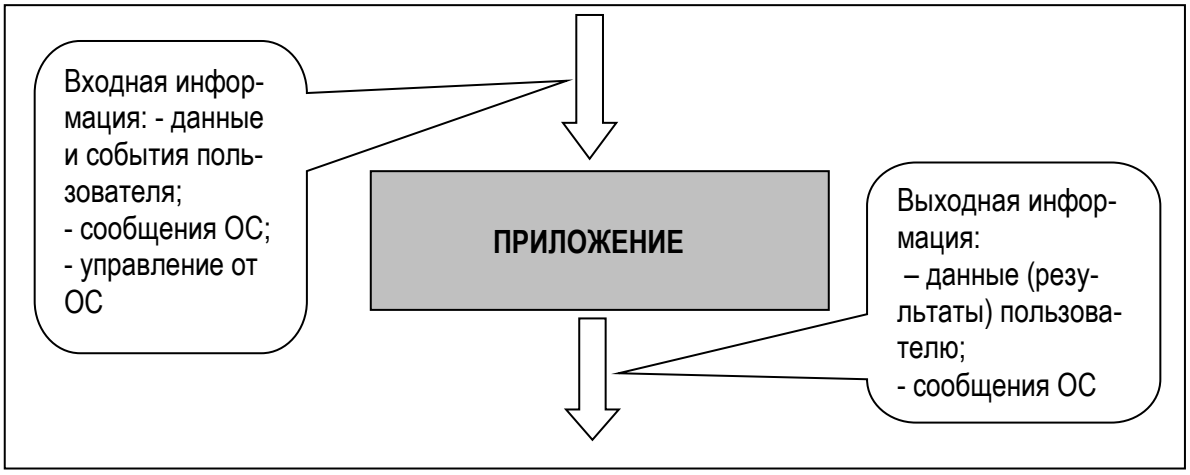


Рисунок 1 - Потоки данных приложения

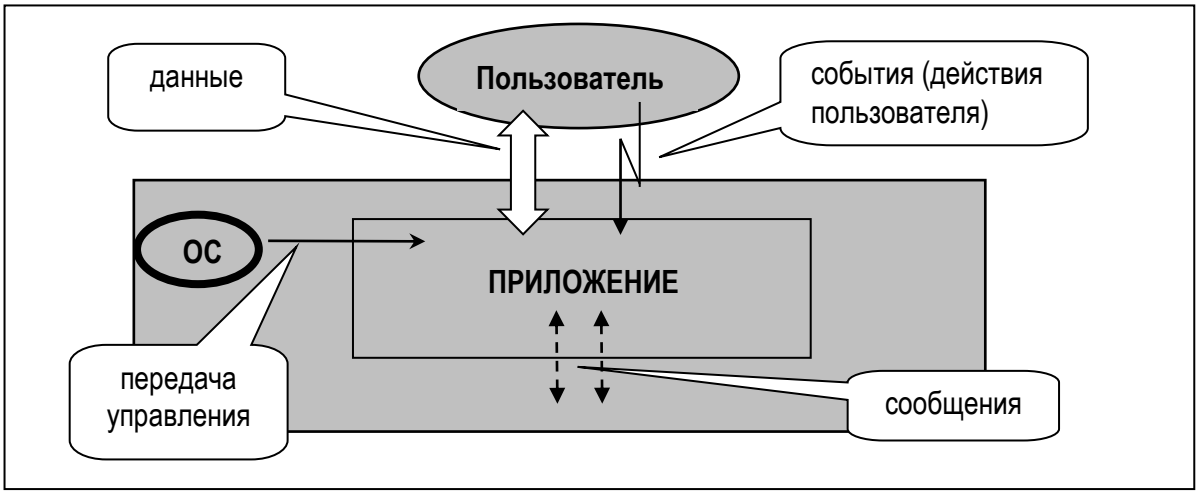


Рисунок 2 - Потоки данных приложения

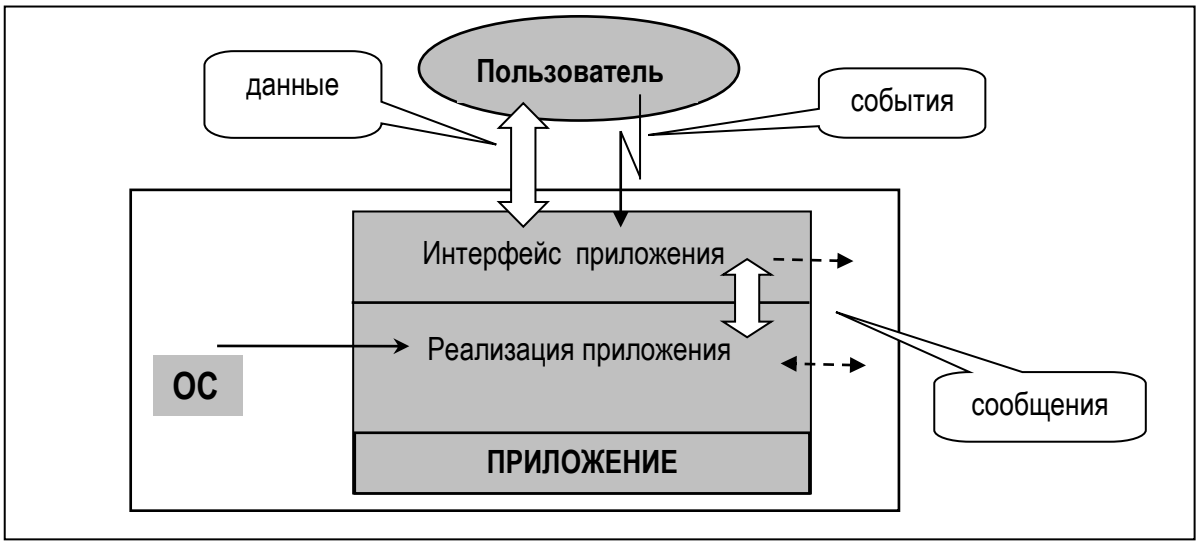


Рисунок 3 - Структура приложения

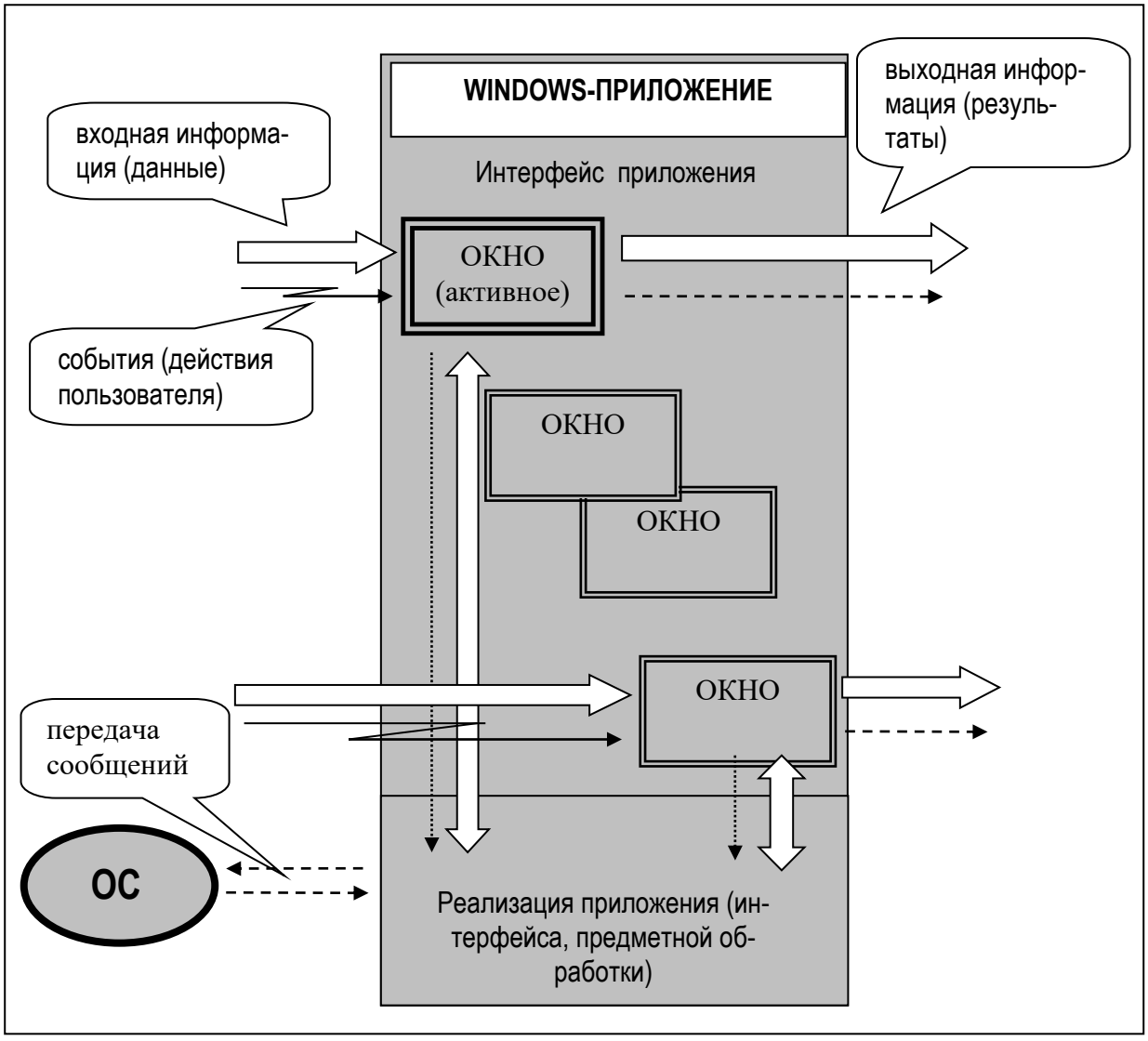


Рисунок 4 - Структура приложения

Более детально Windows-приложение может быть специфицировано в терминах обрабатываемых сообщений. Здесь поведенческое описание может представлять спецификацию сообщений, направляемых приложению, и спецификацию реакций приложения на полученные сообщения (при этом приложение по-прежнему рассматривается как “черный” ящик). Приложения общаются с другими ресурсами, другими приложениями и даже со своими модулями, функциями через посредника – ОС, путем посылки сообщений. Через каждое активное приложение проходит поток сообщений. Приложение циклически отбирает те, к которым оно чувствительно, и организует их обработку (рисунок 5). С учетом оконного интерфейса приложений схема их взаимодействия может быть представлена как на рисунке 6.

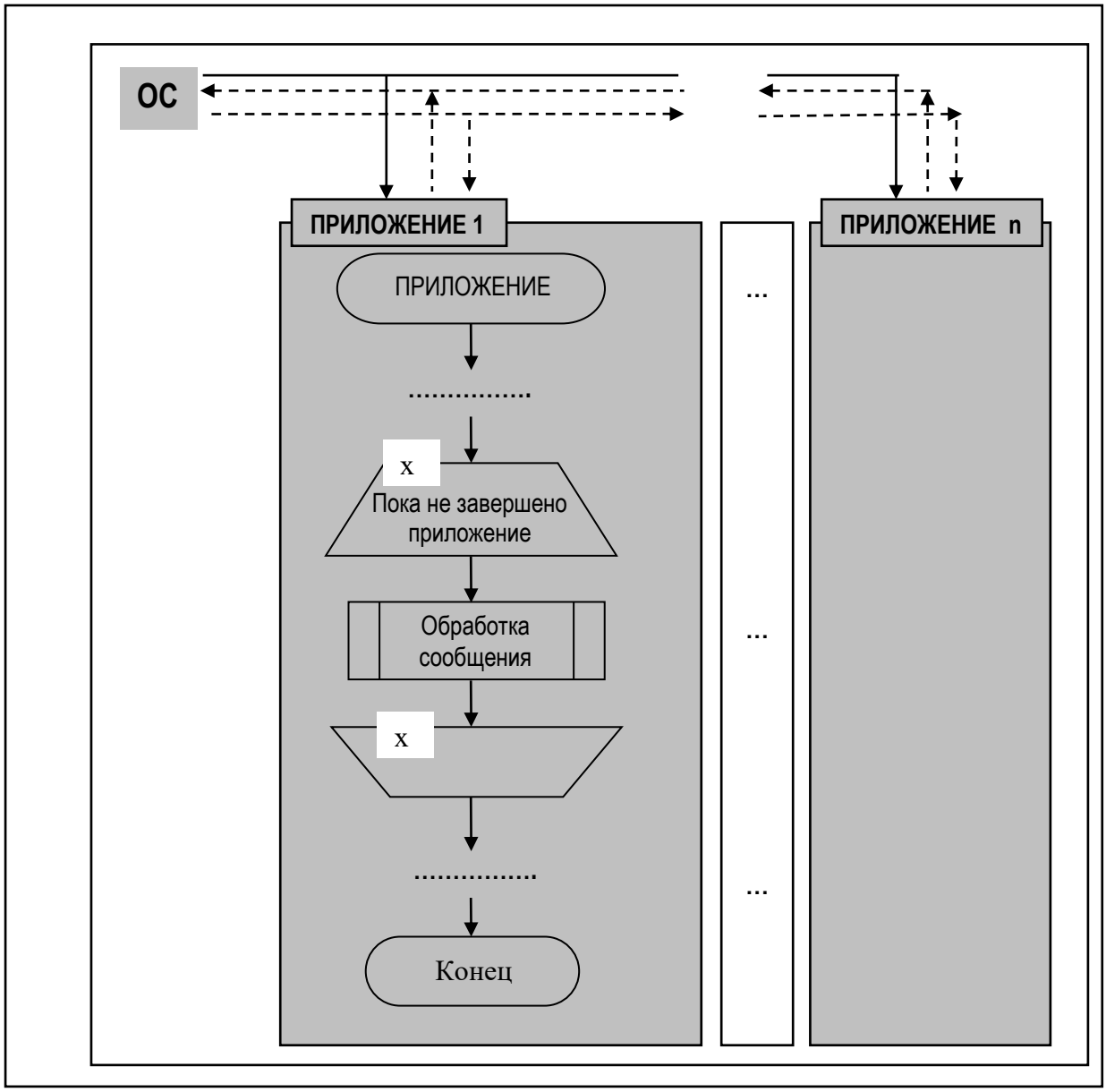


Рисунок 5 - Взаимодействие приложений с ОС

Структура оконных приложений (уровень реализации). С точки зрения программиста (уровень реализации) приложение как набор описаний (текстов) может быть описано в терминах модулей, функций, ресурсов. Это структурное описание (в виде спецификации модулей, схемы иерархии модулей, схемы включения модулей в другие модули), функциональное описание (порядок функционирования приложения на

уровне модулей, представленный, например, в виде графической схемы обобщенного алгоритма работы приложения). Более детально на этом уровне Windows-приложение специфицируется алгоритмически с учетом специфики обрабатываемых сообщений.

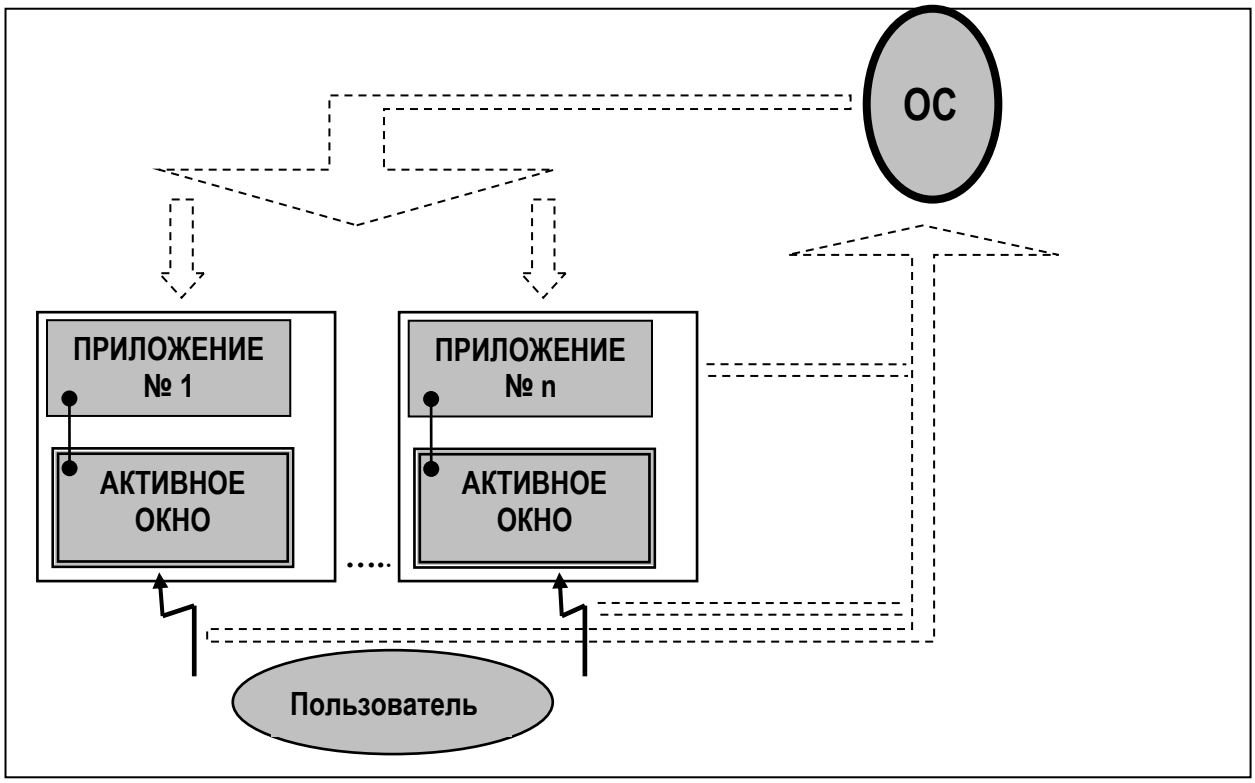


Рисунок 6 - Взаимодействие приложений с ОС

Соответственно оконное приложение можно рассматривать как совокупность описаний - основной программы WinMain (аналога main) и набора типовых комплектов. Типовой комплект компонентов приложения представлен на рисунке (рисунок 7). Он включает описание ресурсов окна (стиля, вида, состава и размещения элементов управления в окне и т.п.) и описание алгоритмов работы программы – обработчика сообщений, адресуемых окну. Оба описания каждого комплекта компонентов связываются при создании приложения. При активизации окно предоставляется пользователю (визуализируется на основе описания ресурсов окна) системой как элемент интерфейса. Пользователь, совершая события – действия над элементами управления окна, над самим окном либо другие действия, непосредственно не связанные с окном в момент его активизации (перемещение или нажатие клавиш “мыши”, манипуляции с клавиатурой и т.п.) посылает соответствующие сообщения, пересылаемые ОС обработчику сообщений, связанному с окном. Указанное иллюстрируется рисунком (рисунок 8). С учетом изложенного структура приложения и состав приложения представлены на рисунках 9, 10.

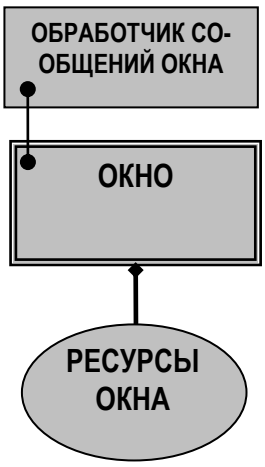


Рисунок 7 - Основные компоненты приложения

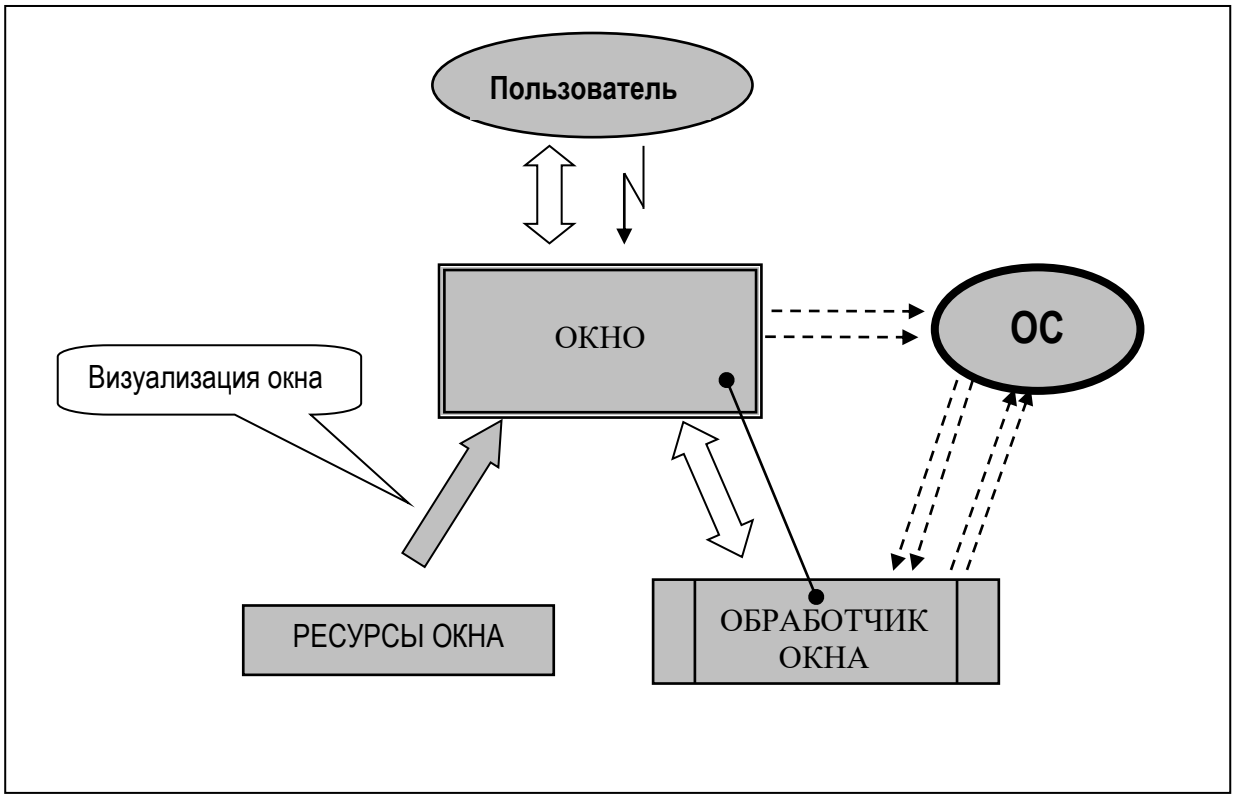


Рисунок 8 - Взаимодействие основных компонентов приложения

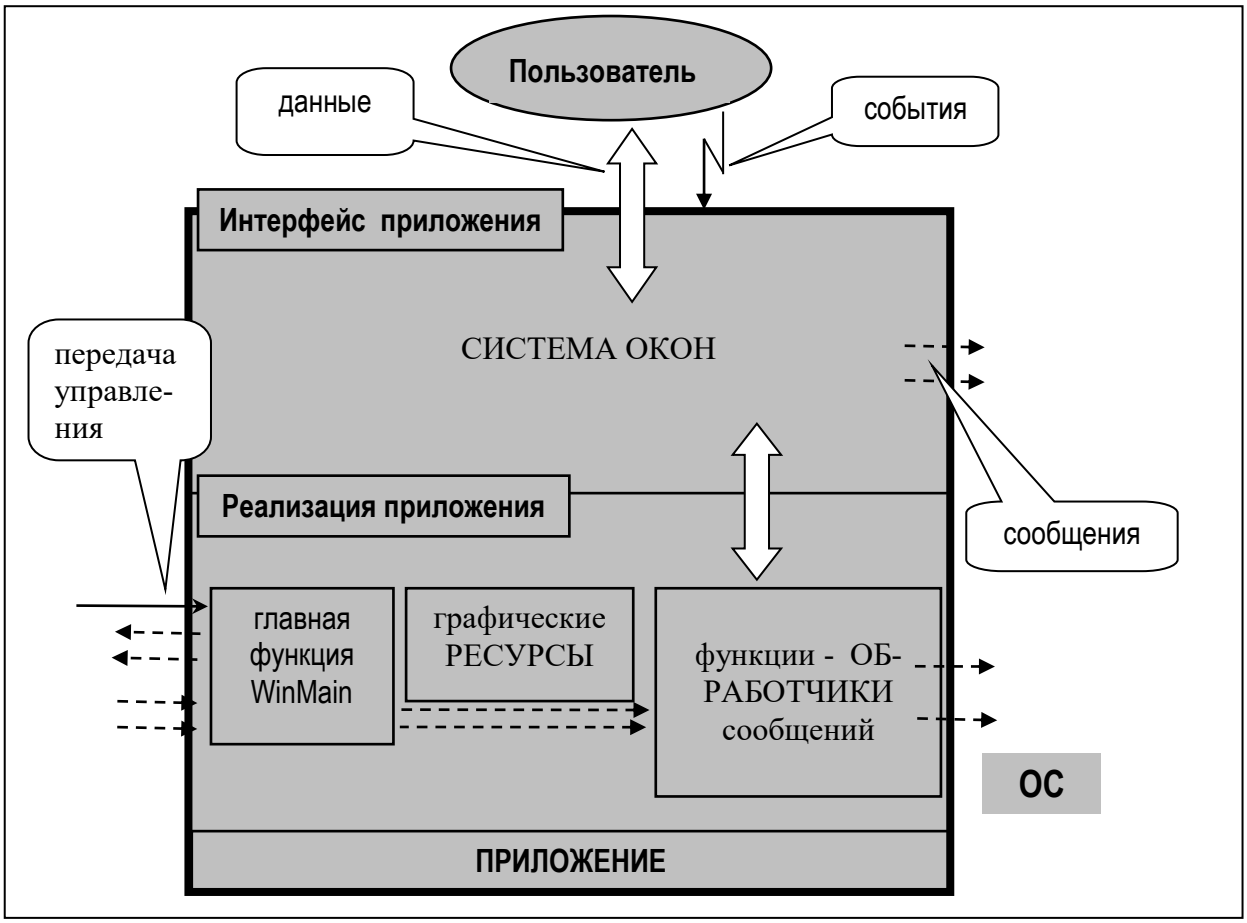


Рисунок 9 - Структура приложения

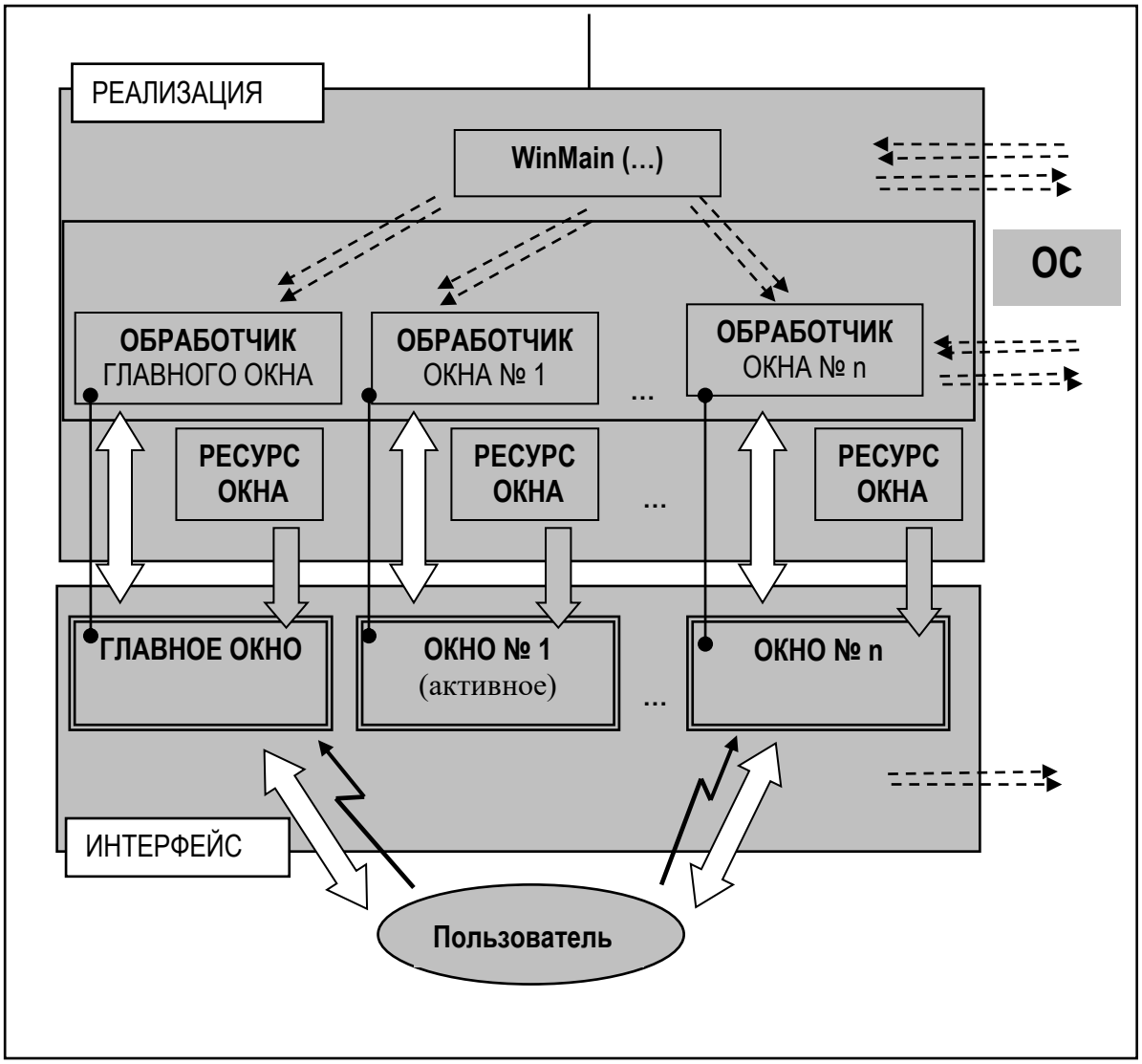


Рисунок 10 - Состав приложения

Примерная структура программ оконного приложения представлена ниже

< Команды_препроцессора >
< Прототип_функции_обработчика_главного_окна >
[< Прототипы_других_функций_обработчиков >]
[< Прототипы_прикладных_функций >]
[< Описания_глобальных_объектов >]
< Описание_главной_функции_WinMain >
< Описание_функции_обработчика_главного_окна >
[< Описание_других_функций_обработчиков >]
[< Описание_прикладных_функций >] .

2 Преобразование типов, ввод-вывод данных

Преобразование типов. Поскольку в оконных приложениях при вводе-выводе большинство функций работает с данными, интерпретируемыми как строковые данные, то пользователю нужно самостоятельно приводить данные к нужному типу.

Для преобразования строки в вещественное значение можно использовать функцию atof - ПРЕОБРАЗОВАТЬ_В_ВЕЩЕСТВЕННОЕ (*Строка*).

Пример использования функции

```
char MyString[20] = “-123.75”;
float MyFloat;
……..
MyFloat = atof (MyString);  .
```

Для преобразования числовых значений в строку можно использовать функцию wsprintf - ПРЕОБРАЗОВАТЬ_В_СТРОКУ (*Строка*, *ШаблонВывода*, *СписокВывода*).

Пример использования функции

```
char szMyString [80] = “ ”;
……..
wsprintf (szMyString, "Был год - %d", 1952);  .
```

Для преобразования вещественного значения в строку можно использовать аналогичную функцию sprintf либо функцию _gcvt ПРЕОБРАЗОВАТЬ_В_СТРОКУ (*ВыводимоеЧисло*, *ДлинаСтроки*, *Строка*) с прототипом

```
char *_gcvt(double Value, int OunputStringLength, char *OunputString)  .
```

Пример использования функций

```
#include <stdio.h>
……
float FloatNumber1 = 12.345;
float FloatNumber2 = -13.345;
char OutputStr[100];
……..
sprintf(OutputStr,"%f",FloatNumber1);
TextOut(hdc, 10,10, 9, strlen(OutputStr));
……
_gcvt (FloatNumber2, strlen(OutputStr), OutputStr );
TextOut(hdc, 100,100, OutputStr, strlen(OutputStr));  .
```

Ввод данных. Ввод данных для обработки в приложении может производиться, например: - из окошек редактирования (элементов управления) диалоговых окон; - из файлов (с использованием функций традиционных и объектно-ориентированных библиотек ввода-вывода C, C++).

Ввод текста. Выполняется, например, с использованием окошка (поля) редактирования ресурса диалоговое окно и функции GetDlgItemText ВВЕСТИ_ТЕКСТ_ИЗ_ПОЛЯ_ДИАЛОГОВОГО_ОКНА (*ДескрипторОкна*, *ДескрипторОкнаРедактирования*, *СтрокаДляВводаДанных*, *ДлинаСтроки*) с прототипом

```
UINT GetDlgItemText (HWND hDlg, int nIDDlgItem, LPTSTR lpString, int nMaxCount)  ,
```

где используются параметры

- *ДескрипторОкна диалога*, содержащего окно редактирования *hDlg;*

- *ДескрипторОкнаРедактирования nIDDlgItem;*
- *СтрокаДляВводаДанных lpString;*
- *ДлинаСтроки nMaxCount.*

Пример использования функции в диалоговом окне с hDlg для записи содержимого окошка редактирования IDC_EDIT1, введенного пользователем, в строку RezultString для дальнейшего использования в приложении

```
GetDlgItemText (hDlg, IDC_EDIT1, RezultString, 15)  .
```

Ввод целых чисел. Выполняется с использованием окошка редактирования ресурса – диалоговое окно и функции GetDlgItemInt ВВЕСТИ_ЦЕЛОЕ_ИЗ_ПОЛЯ_ДИАЛОГОВОГО_ОКНА (*ДескрипторОкна*, *ДескрипторПоляРедактирования*, NULL, bool *НаличиеЗнака*) с прототипом

```
UINT GetDlgItemInt (HWND hDlg, int nIDDlgItem, BOOL *lpTranslated, BOOL bSigned)  ,
```

где используются аналогичные параметры, а параметр *НаличиеЗнака bSigned* управляет преобразованием числа. Если *НаличиеЗнака* = 1, то результат ввода преобразуется в целое со знаком; если *НаличиеЗнака* = 0, то результат ввода преобразуется в целое без знака.

Пример использования функции

```
int    i;
int    j;
……….
i =  GetDlgItemInt (hDlg, IDC_EDIT1, NULL, 0);
j =  GetDlgItemInt (hDlg, IDC_EDIT1, NULL, 1);   .
```

Ввод вещественных чисел. Производится аналогично вводу строковых данных, но с последующим приведением введенной строки к вещественному типу, например, с помощью функции преобразования atof.

Выод данных. Типовые средства вывода данных из приложений включают: - функции вывода данных на экран (в пользовательскую, клиентскую часть окна) типа TextOut, DrawText, вывод сообщений в виде окон сообщений MessageBox и другие; - функции вывода данных в файлы (используются традиционные и объектно-ориентированные библиотеки ввода-вывода C, C++).

Ввод-вывод данных требует указания контекста используемого для этого устройства. Для его хранения используется переменная типа HDC (дескриптор контекста устройства), например, HDC hdc. Так в функции-обработчике дескриптор получают от ОС Windows при выполнении функции BeginPaint, например

```
case WM_PAINT:
     hdc = BeginPaint(hWnd, &ps);
```

и освобождают функцией EndPaint(hWnd, &ps). Могут быть использованы также и другие функции, например, GetDC, ReleaseDC. Здесь hWnd – дескриптор окна, &ps – ссылка на структуру типа PAINTSTRUCT (PAINTSTRUCT ps), которая хранит информацию о клиентской части окна.

Вывод текста. Вывод числовых данных требует предварительного преобразования чисел в строковый формат. Вывод выполняется, например, с использованием функции TextOut ВЫВЕСТИ_ТЕКСТ (*Окно*, *КоординатаX*, *КоординатаY*, *СтрокаВывода*, *ДлинаСтроки*).

Пример использования функции

```
char lpszString[ ] = “Строка вывода”;
int   iStringLength = strlen (lpszString);
int x = 50;
int y = 50;
……..
TextOut (hdc, x, y, lpszString, iStringLength); .
```

Пример использования функции для вывода целых значений от 0 до 10 и их квадратов

```
for (i=0; i<10; i++)
{
     wsprintf ( szMyString, " Значение - %d ", i);
     TextOut ( hdc, 0, 115 + 15 * (i + 1), szMyString, strlen(szMyString ) );
     wsprintf ( szMyString, "его квадрат - %d",  i * i);
     TextOut ( hdc, 140, 115 + 15 * (i + 1), szMyString, strlen(szMyString ) );

};    .
```

Вывод окна сообщения. Производится с использованием функции MessageBox ВЫВЕСТИ_ОКНО_СООБЩЕНИЯ (*Окно*, *СтрокаСообщения*, *НазваниеОкна*, *СоставОкна*). Прототип функции

```
int  MessageBox (HWND hWnd, LPCTSTR  lpText, LPCTSTR  lpCaption, UINT  MBType).
```

Параметры:

- *СтрокаСообщения* - указатель на строку С (с нуль-символом), которая содержит выводимое сообщение;
- *НазваниеОкнаСообщения* - указатель на строку С (с нуль-символом), которая содержит заголовок окна сообщения. Если указатель равен NULL, то по умолчанию используется заголовок “Error”;
- *СоставОкна* - определяет состав окна сообщения (комбинацию пиктограмм и командных кнопок) и его поведение. Используемые пиктограммы: MB_ICONHAND, MB_ICONSTOP, MB_ICONERROR, MB_ICONQUESTION, MB_ICONEXCLAMATION, MB_ICONWARNING, MB_ICON-ASTERISK, MB_ICONINFORMATION. Используемые комбинации кнопок: MB_ABORTRETRYIGNORE, MB_OK, MB_OKCANCEL, MB_YESNO, MB_RETRYCANCEL, MB_YESNOCANCEL.

Основные значения параметров MessageBox, а также типы возвращаемого результата приведены в таблицах - опции функции MessageBox (таблицы 4-6).

Таблица 4 - Наиболее распространенные значения параметра MBType

Значение	**Результат**
MB_ABORTRETRYIGNORE	отображаются кнопки Abort, Retry, Ignore
MB_ICONEXCLAMATION	отображается пиктограмма с восклицательным знаком

MB_ICONINFORMATION	отображается информационная пиктограмма с буквой "I"
MB_HAND	отображается пиктограмма с надписью "Stop"
MB_ICONQUESTION	отображается пиктограмма с вопросительным знаком
MB_ICONSTOP	то же, что и MB_HAND
MB_OK	отображается кнопка OK
MB_OKCANCEL	отображаются кнопки OK, Cancel
MB_RETRYCANCEL	отображаются кнопки Retry, Cancel
MB_YESNO	отображаются кнопки Yes, No
MB_YESNOCANCEL	отображаются кнопки Yes, No, Cancel

Таблица 5 - Встроенные пиктограммы

Константа	**Вид пиктограммы**
IDI_APPLICATION	стандартная системная пиктограмма
IDI_ASTERISK	информационная пиктограмма с буквой "I"
IDI_EXCLAMATION	пиктограмма с восклицательным знаком
IDI_HAND	пиктограмма с надписью "Stop"
IDI_QUESTION	пиктограмма с вопросительным знаком
IDI_WINLOGO	пиктограмма с логотипом Windows

Таблица 6 - Возвращаемые значения

Нажатая кнопка	**Возвращаемое значение**
Abort	IDABORT
Retry	IDRETRY
Ignore	IDIGNORE
Cancel	IDCANCEL
No	IDNO
Yes	IDYES
OK	IDOK

Пример использования функции

MessageBox (hWnd, “Пример ”, "Демо", MB_OK); .

Окно сообщения можно использовать для ввода управляющей информации в виде данных о нажатых кнопках, например

```
if   ( MessageBox (hWnd, " Пример ", "Демо", MB_OKCANCEL) == IDCANCEL)
MessageBox (hWnd, "IDCANCEL", "Демо", MB_OKCANCEL);  .
```

Вывод текста с усиленными возможностями редактирования функцией DrawText ИЗОБРАЗИТЬ_ТЕКСТ (*ДескрипторУстройства*, *АдресСтрокиВывода*, *ДлинаСтроки-Вывода*, *АдресСтруктурыПрямоугольникаВывода*, *ФлагиФорматирования*), где *АдресСтруктурыПрямоугольникаВывода* задает прямоугольную часть клиентского окна (описывает ее в структуре типа RECT) для организации вывода. Прототип

int DrawText (HDC *hDC*, LPCTSTR *lpString*, int *nCount*, LPRECT *lpRect*, UINT *uFormat*) .

Пример использования функции

```
RECT rt;
rt.left = 5;  rt.top = 5;  rt.right = 1000;  rt.bottom = 1000;
……..
GetClientRect (hWnd, &rt);
DrawText (hdc, “Hello”, strlen(“Hello”), &rt, DT_CENTER);  .
```

Лекция № 4 Сообщения

1 Сообщения. Обработчики сообщений

Через операционную систему Windows проходят и направляются на обработку в приложения разнообразные сообщения о событиях, происходящих в программах, устройствах и т.д. Синхронные сообщения, адресуемые приложению и соблюдающие порядок, очередь обработки, образуют очередь сообщений. Асинхронные сообщения, адресуемые приложению (сообщения таймера, перерисовки, завершения работы и др.), обслуживаются вне очереди. Каждое сообщение сопровождается рядом атрибутов, исчерпывающе характеризующих сообщение и необходимых для его обработки (например, системное время момента наступления события, состояние клавиатуры, “мыши”, идентификация источника сообщения и т.д.).

Сообщение, адресуемое приложению, предварительно обрабатывается фрагментом главной функции WinMain с целью отправки на исполнение в соответствующие функции-обработчики. Основные сообщения, их названия и прототипы функций для их обработки приведены ниже (таблица 7).

Таблица 7 - Прототипы функций обработки сообщений

Сообщение	**Прототип функции обработки сообщения**
WM_CHAR	void Cls_OnChar(HWND hwnd, UINT ch, int cRepeat)
WM_COMMAND	void Cls_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify)
WM_CREATE	BOOL Cls_OnCreate(HWND hwnd, CREATESTRUCT FAR* lpCreateStruct)
WM_DESTROY	void Cls_OnDestroy(HWND hwnd)
WM_GETMINMAXINFO	void Cls_ OnGetMinMaxInfo(HWND hwnd, MINMAXINFO FAR* lpMinMaxInfo)
WM_NITDIALOG	BOOL Cls_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam)
WM_KEYDOWN	void Cls_OnKeyDown(HWND hwnd, UINT vk, BOOL fDown, int cRepeat, UINT flags)
WM_KEYUP	void Cls_OnKeyUp(HWND hwnd, UINT vk, BOOL fDown, int cRepeat, UINT flags)
WM_KILLFOCUS	void Cls_OnKillFocus(HWND hwnd, HWND hwndNewFocus)
WM_LBUTTONDOWN, WM_LBUTTONDBLCLK	void Cls_OnLButtonDown(HWND hwnd, BOOL fDoubleClick, int x, int y, UINT keyFlags)

WM_LBUTTONUP	void Cls_OnLButtonUp(HWND hwnd, int x, int y, UINT keyFlags)
WM_MOUSEMOVE	void Cls_OnMouseMove(HWND hwnd, int x, int y, UINT keyFlags)
WM_NOTIFY	BOOL Cls_OnNotify(HWND hwnd, INT idCtrl, NMHDR* pnmh)
WM_PAINT	void Cls_OnPaint(HWND hwnd)
WM_QUIT	void Cls_OnQuit(HWND hwnd, int exitCode)
WM_RBUTTONDOWN, WM_RBUTTONDBLCLK	void Cls_OnRButtonDown(HWND hwnd, BOOL fDoubleClick, int x, int y, UINT keyFlags)
WM_RBUTTONUP	void Cls_OnRButtonUp(HWND hwnd, int x, int y, UINT flags)
WM_SETCURSOR	BOOL Cls_OnSetCursor(HWND hwnd, HWND hwndCursor, UINT codeHitTest, UINT msg)
WM_SETFOCUS	void Cls_OnSetFocus(HWND hwnd, HWND hwndOldFocus)
WM_SHOWWINDOW	void Cls_OnShowWindow(HWND hwnd, BOOL fShow, UINT status)
WM_SIZE	void Cls_OnSize(HWND hwnd, UINT state, int cx, int cy)
WM_SYSCHAR	void Cls_OnSysChar(HWND hwnd, UINT ch, int cRepeat)
WM_SYSCOMMAND	void Cls_OnSysCommand(HWND hwnd, UINT cmd, int x, int y)
WM_SYSKEY	void Cls_OnSysKey(HWND hwnd, UINT vk, BOOL fDown, int cRepeat, UINT flags)
WM_TIMER	void Cls_OnTimer(HWND hwnd, UINT id)

Общая схема обработки сообщений. Сообщение – уведомление приложения о том, что произошло событие, требующее обработки, реакции приложения. Сообщения приходят от пользователей (например, при выборе пункта меню, нажатии кнопки и т.п.), от других приложений, от самого приложения (например, посылкой сообщения на перерисовку клиентской области окна), от ОС (например, сообщения таймера). Каждое сообщение сопровождается рядом атрибутов (например, системное время, состояние клавиатуры, мыши, идентификация источника сообщения и т.д.), исчерпывающе характеризующих сообщение для его обработки.

Сообщения обозначаются как WM_SIZE, WM_MOVE, WM_CLOSE, WM_COMMAND, WM_LBUTTONDOWN, WM_DESTROY, WM_QUIT, WM_KEYDOWN, WM_TIMER, WM_CHAR, WM_CUT, WM_COPY и т.п. Непосредственно с приложением, окном, клиентской областью связаны сообщения WM_PAINT (перерисовать окно при изменении его размеров), WM_CLOSE при свертывании клиентского окна, WM_DESTROY при его закрытии, WM_QUIT при закрытии приложения.

Предварительная обработка сообщений с целью их отправки в соответствующие обработчики выполняется фрагментом WinMain, представленным ниже, а общая схема обработки сообщений иллюстрируются рисунком 11.

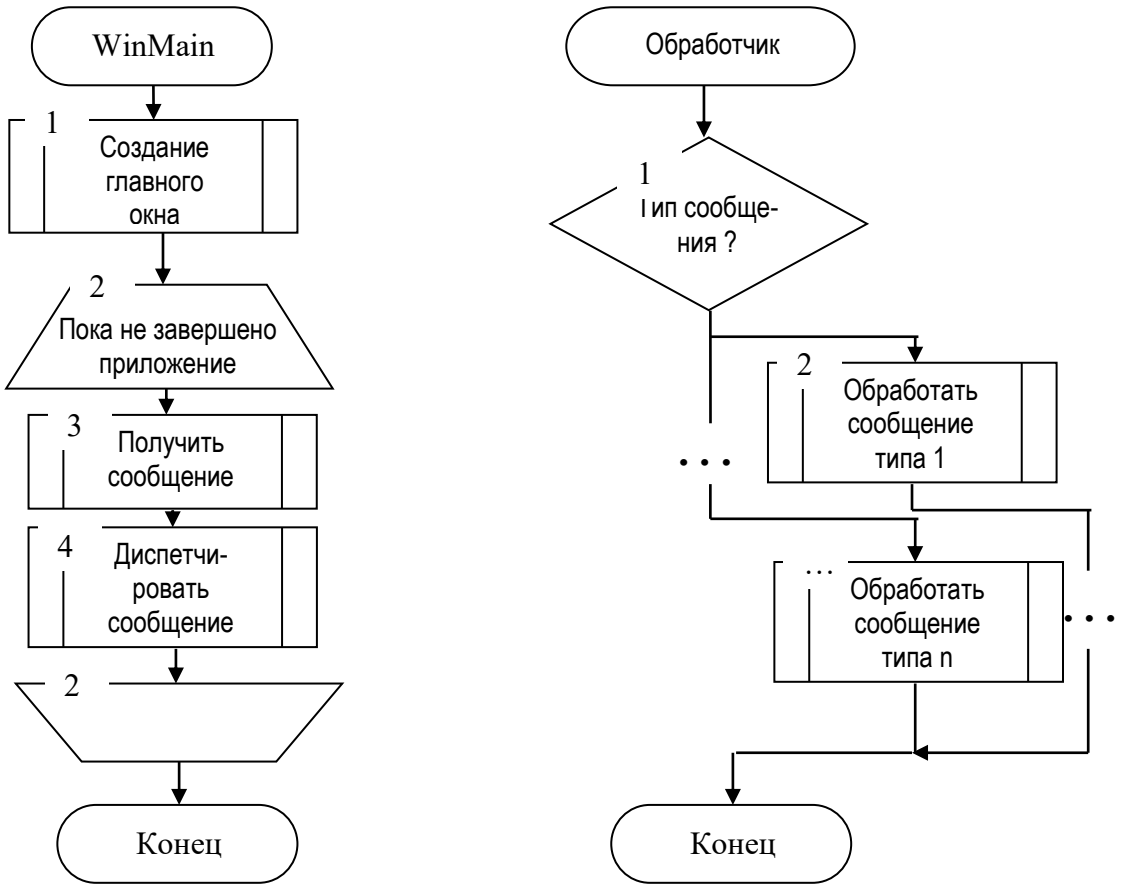


Рисунок 11 - Алгоритмы работы функций ТКП

```
while (GetMessage (&lpMsg, NULL, … , … ))
{
    TranslateMessage (&lpMsg);
    DispatchMessage (&lpMsg);
}      .
```

2 Функция-обработчик сообщений главного окна ТКП

Назначение функции – обработка сообщений, адресуемых окну. Соответственно обработчик реагирует на сообщения.

Основные сообщения: – при визуализации окна принимается сообщение WM_CREATE; - при изменении размеров окна принимается сообщение WM_PAINT с требованием перерисовки окна; – при завершении работы приложения принимается сообщение WM_QUIT; - закрытие окна сопровождается сообщением WM_DESTROY, что при закрытии главного окна означает, как правило, и закрытие приложения. Соответственно при обработке сообщения WM_DESTROY системе надо переслать сообщение WM_QUIT; - сообщение WM_COMMAND о выборе пункта пользовательского меню окна приложения и другие.

Функции-обработчики сообщений окон приложения, включая главное и диалоговые окна, имеют общий прототип

LRESULT CALLBACK <ИмяОбработчикаОкна> (HWND hWnd, UINT Message, WPARAM wParam, LPARAM lParam); .

Здесь
- HWND hWnd - <ДескрипторОкна> или <ДескрипторДиалоговогоОкна >, чьи сообщения обрабатываются;
- UINT Message - <ТипСообщения>, которое обрабатывается;
- WPARAM wParam - < ПараметрыСообщения >, которое обрабатывается;
- LPARAM lParam - < РасширенныеПараметрыСообщения >, которое обрабатывается.

Для передачи параметров используются типы, определенные как

```
typedef  UINT   WPARAM;
typedef LONG LPARAM;  .
```

Тип результата работы функции-обработчика окна (главного окна) определен как

```
typedef LONG LRESULT; .
```

Указатели на соответствующую функцию-обработчик описаны как

```
typedef  LRESULT ( CALLBACK*  WNDPROC ) (HWND, UINT, WPARAM, LPARAM);
typedef   BOOL     ( CALLBACK*  DLGPROC ) (HWND, UINT, WPARAM, LPARAM);  .
```

```
LRESULT CALLBACK
<ИмяОбработчикаОкна> (HWND      <ДескрипторОкна >,
                      UINT      <ТипСообщения>,
                      WPARAM <ПараметрыСообщения>,
                      LPARAM   < РасширенныеПараметрыСообщения >)
```

Например, прототип функции-обработчика главного окна WndProc

```
LRESULT CALLBACK  WndProc (HWND, UINT, WPARAM, LPARAM);
```

и описание функции-обработчика

```
LRESULT CALLBACK WndProc (HWND hWnd, UINT Message,
                          WPARAM wParam, LPARAM lParam)
{
    HDC hdc;
    PAINTSTRUCT ps;
    switch (Message)
    {
        case WM_PAINT:
            hdc = BeginPaint(hWnd, &ps);
            < Фрагмент_пользователя >
            ValidateRect(hWnd,NULL);
            EndPaint(hWnd, &ps);
            break;
        case WM_DESTROY:
            PostQuitMessage(0);
```

```
                break;
        default:
                return (DefWindowProc(hWnd, Message, wParam, lParam));
}
return 0;  .
```

Лекция № 5 Типовой каркас оконного windows-приложения

1 Структура ТКП

Ниже рассмотрена базовая структура оконных приложений, называемая типовым каркасом приложения (ТКП). Реальное приложение создается путем до программирования и, или модификации типового каркаса. структура ТКП представлена ниже (рисунок 12).

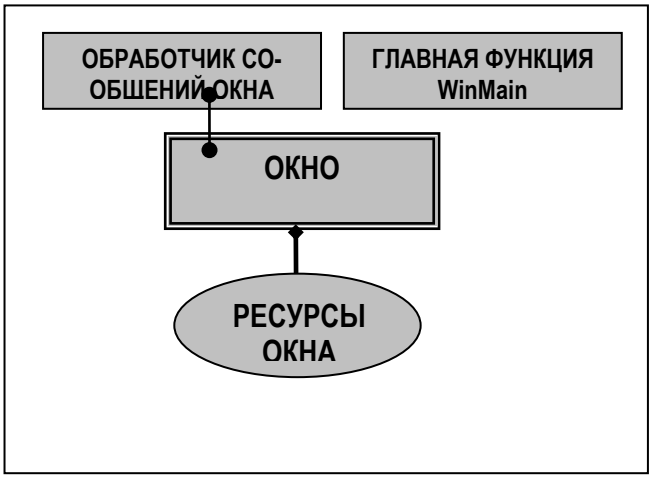


Рисунок 12 - Структура ТКП

Прохождение потоков данных в ТКП представлено на рисунке 13.

Структура программного обеспечения ТКП. ТКП содержит две основные функции – главную WinMain (аналог main) и обработчик сообщений окна приложения. Они связаны друг с другом ассоциативно (обслуживают одно приложение) и информационно через общее главное окно интерфейса. Главная его создает и визуализирует, передавая активность и управление, и обеспечивая пользователя инструментом взаимодействия с приложением. Функция-обработчик (функция окна, оконная функция) относится к функциям обратного вызова, инициируемых ОС посылкой сообщений с уточняющими параметрами. Обработчик, обрабатывая сообщения окну, реализует нужные пользователю алгоритмы. С точки зрения классического управления в схеме иерархии эти модули стоят на одном уровне, так как не запускают друг друга непосредственно. Примерная структура ТКП оконного приложения представлена ниже, а алгоритмы указанных функций представлены на рисунке (рисунок 14).

```
< Команды_препроцессора >
< Прототип_функции_обработчика_главного_окна >
 [ < Описания_глобальных_объектов > ]
< Описание_главной_функции_WinMain >
< Описание_функции_обработчика_главного_окна >   .
```

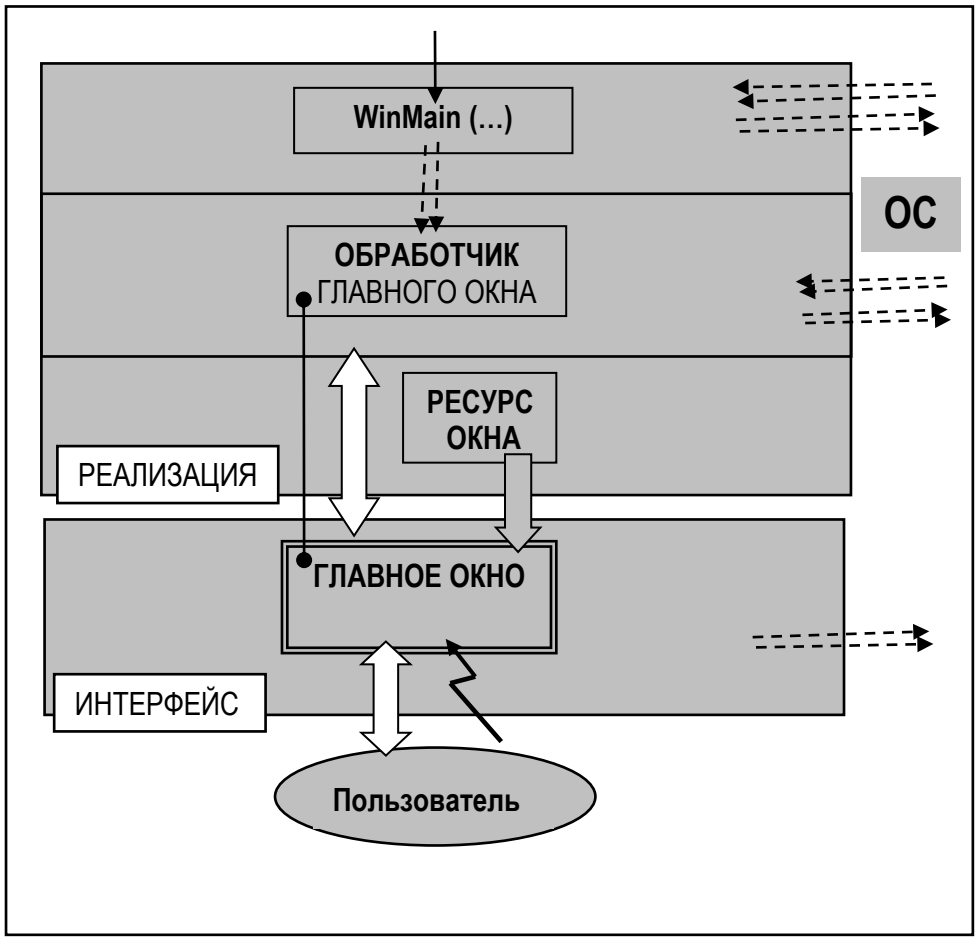


Рисунок 13 - Потоки данных ТКП

Ниже представлена структура функции WinMain, вызываемой из ОС,

СОЗДАНИЕ-ГЛАВНОГО-ОКНА-ПРИЛОЖЕНИЯ
    Описание стиля главного окна
    Регистрация стиля окна
    Создание экземпляра окна
    Визуализация окна
    Обновление окна
РАБОТА-С-СООБЩЕНИЯМИ
    ЦИКЛ-ПОКА не завершено приложение (не поступило сообщение WM_QUIT)
        Получить очередное сообщение из очереди сообщений приложения
        Диспетчировать сообщение (передать обработчику соответствующего окна)
    КОНЕЦ-ЦИКЛА
Завершение работы приложения .

Информационная модель ТКП. С информационной точки зрения приложение может рассматриваться как совокупность специальных структур данных, обеспечивающих его работу. Работа ТКП базируется на использовании структур типа WNDCLASS, PAINTSTRUCT, HDC, MSG и других. WNDCLASS - структура для хранения информации о стиле окна. Описывается пользователем и после регистрации стиля окна в ОС <ИмяСтиляОкна> может использоваться для создания конкретных экземпляров окна. Описание полей структуры WNDCLASS.

```
typedef struct tagWNDCLASS
{
    LPCSTR lpszClassName;  HINSNANCE hInstance;  WNDPROC lpfnWndProc;
    HCURSOR hCursor;  HICON hIcon;
    LPCSTR lpszMenuName;  HBRUSH hbrBackground;
    UINT Style;
    cbClsExtra;
    cbWndExtra;
}   WNDCLASS.
```

Здесь

- lpszClassName = <ИмяСтиляОкна> - имя класса, стиля окна;
- hInstance =<ДескрипторПриложения> - дескриптор приложения, которое регистрирует класс окна (берется из значения соответствующего параметра функции WinMain, которое задается ОС при запуске приложения);
- lpfnWndProc = <ИмяОбработчикаОкна> - значение указателя на функцию-обработчик сообщений окна, которая выполняет все задачи, связанные с окном (функция описывается пользователем);

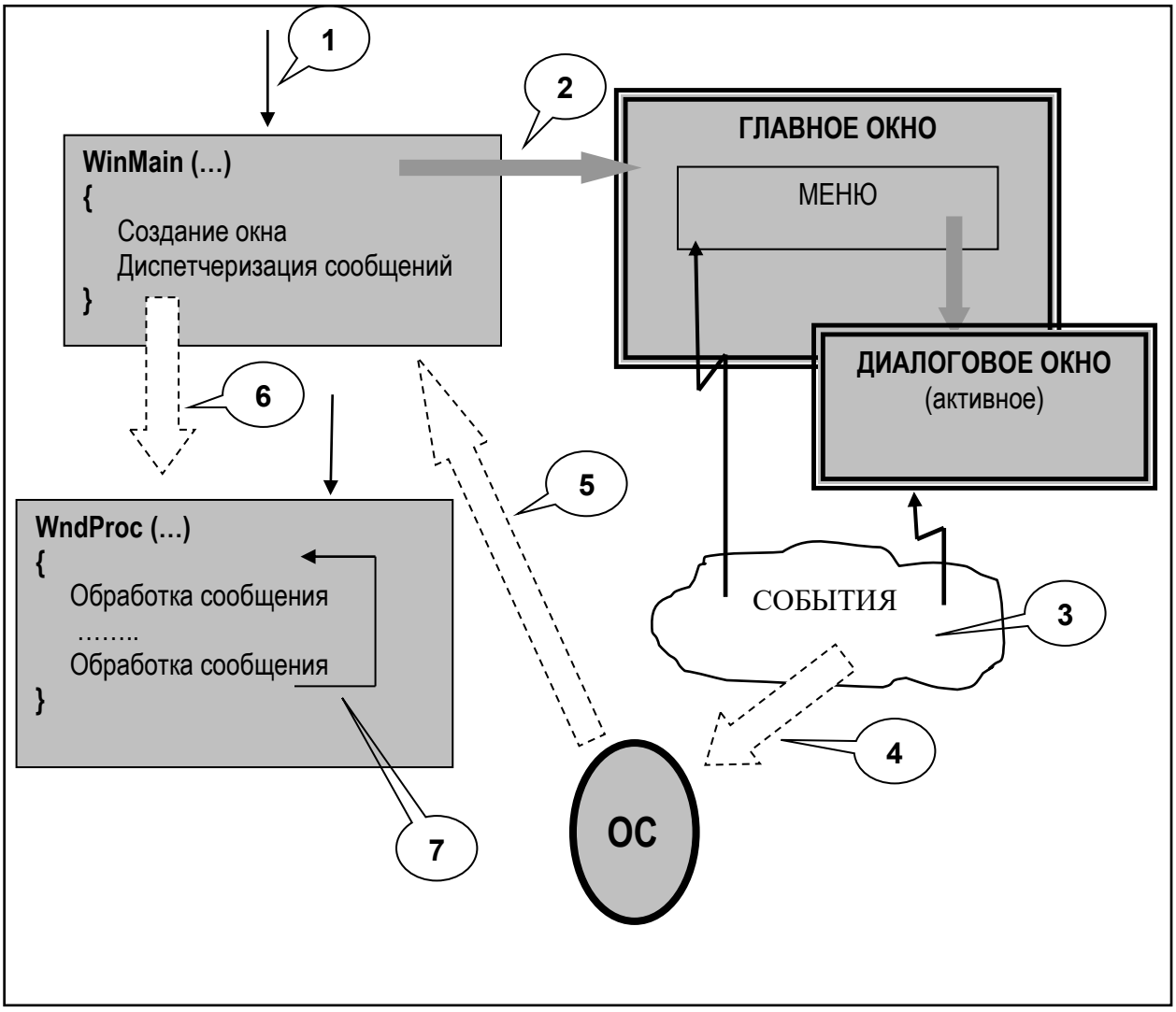


Рисунок 14 - Схема обработки сообщений в ТКП

hCursor – ресурс окна, тип курсора окна;

hIcon - ресурс окна, задает вид пиктограммы при выводе окна в свернутом виде (в виде пиктограммы, обычно NULL);

lpszMenuName = <ИмяРесурсаМеню> – ресурс окна - пользовательское меню;

hbrBackground - ресурс окна, определяет кисть, используемую для закраски фона окна (значением данного параметра может быть как идентификатор физической кисти, так и значение цвета);

style – доопределяет класс, стиль окна (например, параметр CS_VREDRAW обеспечивает перерисовку содержимого клиентской области окна при изменении размера окна по вертикали и т.п.);

cbClsExtra - задает количество дополнительных байт, выделяемых структуре WNDCLASS (обычно NULL);

cbWndExtra - задается количество дополнительных байт, выделяемых для всех дополнительных структур, которые создаются с использованием данного класса, стиля окна (обычно NULL). Использование структуры иллюстрируется ниже

```
char   szWindowClass [] = <ИмяСтиляОкна> ;
……..
WNDCLASS      <СтруктураДанныхОкна>;
<СтруктураДанныхОкна>.lpszClassName   = (LPCSTR) <ИмяСтиляОкна>;
<СтруктураДанныхОкна>.hInstance          = (HINSNANCE) <ДескрипторПриложения>;
<СтруктураДанныхОкна>.lpfnWndProc      = (WNDPROC) <ИмяОбработчикаОкна>;
 <СтруктураДанныхОкна>.hCursor          = (HCURSOR) <ТипКурсора>
                                              = LoadCursor  (NULL, IDC_ARROW);
<СтруктураДанныхОкна>.hIcon              = (HICON) <Пиктограмма>
                                                  = 0;
<СтруктураДанныхОкна>.lpszMenuName    = (LPCSTR) <ИмяРесурсаМеню>;
<СтруктураДанныхОкна>.hbrBackground    = (HBRUSH) <КистьФона>
                                                    = GetStockObject (WHITE_BRUSH);
<СтруктураДанныхОкна>.style              = (UINT) <ДополнительныйСтиль1>
                                                  = CS_HREDRAW|CS_VREDRAW;
<СтруктураДанныхОкна>.cbClsExtra         = (UINT) <ЧислоДополнительныхБайт>
                                                  = 0;
<СтруктураДанныхОкна>.cbWndExtra        = (UINT) <ЧислоДополнительныхБайт>
                                                   = 0;
```

MSG - структура данных для хранения информации о сообщении

```
typedef struct tagMSG
{
     HWND hWnd;
      UINT   message;
      WPARAM  wParam;
      LPARAM   lParam;
      DWORD time;
      POINT pt;
 }    MSG.
```

Здесь

- HWND hWnd - <ДескрипторОкна>;
- UINT   message – код <ТипСообщения>;

- WPARAM wParam <ПараметрыСообщения> (ПервыйПараметрСообщения);
- LPARAM lParam <РасширенныеПараметрыСообщения> (ВторойПараметрСообщения);
- DWORD time время возникновения сообщения;
- POINT pt позиция курсора мыши.

PAINTSTRUCT (см. winuser.h) - структура данных (заполняется каждый раз, когда приложение перехватывает сообщение WM_PAINT) для хранения информации о клиентской области окна, с которой связан контекст устройства КУ. КУ – системный ресурс, (область памяти) для хранения атрибутов объектов, связанных с рисованием (информация о кисти, перьях, шрифтах и т.п.). Используется графическими функциями GDI через <ДескрипторКУ>. При этом предполагается последовательность действий: Получить КУ. Изменить атрибуты КУ. Использовать функции GDI (рисовать). Вернуть КУ. Описание

```
typedef struct tagPAINTSTRUCT
{
     HDC hdc;
     BOOL fErase;
     RECT rcPaint;
     BOOL fRestore;
     BOOL fIncUpdate;
     BYTE rgbReserved [32];
}   PAINTSTRUCT.
```

Здесь
- HDC hdc - <ДескрипторКУ>;
- BOOL fErase - флаг перерисовки фона окна (0), иначе используется прозрачное окно;
- RECT rcPaint – описание прямоугольной области окна, той части, которую надо перерисовать;
- RECT - структура данных для описания прямоугольной области, части окна, которую надо перерисовать

```
typedef struct tagRECT
{
   LONG   left;
   LONG   top;
   LONG   right;
   LONG   bottom;
}  RECT, *PRECT, NEAR *NPRECT, FAR *LPRECT;
```

Пример использования для подготовки рисования

```
PAINTSTRUCT  ps;
ps.rcPaint = { 0, 0 , 290, 73 };
HDC hdc = BeginPaint (hWnd, &ps);  .
```

TEXTMETRIC (см. winuser.h) - информационная структура данных, которая хранит атрибуты (метрики) текущего шрифта, связанного с КУ. Задание атрибутов выполняется функциями SetTextColor(), SetBkColor() и др. Поле tmExternalLeading определяет расстояние между строками, поле tmHeight задает полную высоту символов используемого шрифта и т.д.

```
typedef struct tagTEXTMETRICA
{
```

```
    LONG       tmHeight; Полная высота символов шрифта
    LONG       tmAscent;
    LONG       tmDescent;
    LONG       tmInternalLeading;
    LONG       tmExternalLeading; Расстояние между строками
    LONG       tmAveCharWidth;
    LONG       tmMaxCharWidth;
    LONG       tmWeight;
    LONG       tmOverhang;
    LONG       tmDigitizedAspectX;
    LONG       tmDigitizedAspectY;
    BYTE       tmFirstChar;
    BYTE       tmLastChar;
    BYTE       tmDefaultChar;
    BYTE       tmBreakChar;
    BYTE       tmItalic;
    BYTE       tmUnderlined;
    BYTE       tmStruckOut;
    BYTE       tmPitchAndFamily;
    BYTE       tmCharSet;
}  TEXTMETRICA, *PTEXTMETRICA,  FAR *LPTEXTMETRICA;  .
```

Главная функция ТКП. Прототип главной функции приложения

```
int  WINAPI  WinMain  ( HINSTANCE hInstance,
                        HINSTANCE hPrevInstance,
                        LPSTR lpCmdLine,
                        int nCmdShow); .
```

Здесь

- HINSTANCE hInstance – идентификатор <ДескрипторПриложения> текущего экземпляра приложения (определяется ОС);
- HINSTANCE hPrevInstance - всегда NULL;
- LPSTR lpszCmdLine - указатель на командную строку приложения;
- int nCmdShow - способ <ИзображенияОкнаПриПервомВыводе> (например, параметр SW_SHOWDEFAULT активизирует окно и выводит его с использованием текущих параметров – по умолчанию; параметр SW_SHOWNORMAL активизирует окно и если окно было увеличено или уменьшено до пиктограммы, то оно восстанавливается с учетом начального положения и размера окна).

```
int  WINAPI  WinMain  ( HINSTANCE <ДескрипторПриложения>,
                        HINSTANCE hPrevInstance,
                        LPSTR lpCmdLine,
                        int <ИзображениеОкнаПриПервомВыводе> );
```

Примерный текст типового каркаса пользователя приведен ниже

```
#include <windows.h>
LRESULT CALLBACK    <ИмяОбработчика> (HWND, UINT, WPARAM, LPARAM);
char   szWindowClass [ ] = <ПользовательскоеИмяСтиляОкна>;

int  WINAPI  WinMain (HINSTANCE <ДескрипторПриложения>,
```

```
                        HINSTANCE hPreInstance,
                        LPSTR lpszCmdLine, int   nCmdShow)
{
    < Описание_главной_функции >
}

LRESULT CALLBACK  <ИмяОбработчика> (HWND <ДескрипторОкна>,
                                    UINT  <КодСообщения>,
                                    WPARAM wParam, LPARAM lParam)
{
     < Описание_обработки_сообщений_главного_окна >
}
```

Более детальный текст ТКП приведен ниже

```
 #include <windows.h>
 LRESULT CALLBACK  <ИмяОбработчика> (HWND, UINT, WPARAM, LPARAM);
 char   szWindowClass [ ] = <ПользовательскоеИмяСтиляОкна>;

 int  WINAPI  WinMain (HINSTANCE <ДескрипторПриложения>,
                       HINSTANCE hPreInstance,
                       LPSTR lpszCmdLine, int   nCmdShow)
 {
        СОЗДАНИЕ-ГЛАВНОГО-ОКНА-ПРИЛОЖЕНИЯ
        РАБОТА-С-СООБЩЕНИЯМИ
          MSG       lpMessage;
          while (GetMessage (&lpMessage, NULL, 0, 0))
          {
            TranslateMessage (&lpMessage);
            DispatchMessage (&lpMessage);
          }
  }  .
```

Здесь полученное приложением сообщение копируется в структуру данных типа MSG, на которую указывает адрес lpMsg, и передает его далее на обработку соответствующей функции-обработчику окна

```
 LRESULT CALLBACK  <ИмяОбработчика> (HWND <ДескрипторОкна>,
                                     UINT  <КодСообщения>,
                                     WPARAM wParam, LPARAM lParam)
 {
    HDC   <ДескрипторКонтекстаУстройства>;
    PAINTSTRUCT  <СтруктураКлиентскойОбластиОкна>;
    switch   (<КодСообщения>)
    {
       case WM_PAINT:
           <ДескрипторКонтекстаУстройства>=BeginPaint(<ДескрипторОкна>,
                  &ps);
```

```
            < Фрагмент_пользователя >
            ValidateRect (<ДескрипторОкна>, NULL);
            EndPaint (<ДескрипторОкна>, <СтруктураКлиентскойОбластиОкна>);
            break;
        case <ЗначениеКодаСообщения> :
              < Обработка_сообщения >
        case <ЗначениеКодаСообщения> :
              < Обработка_сообщения >
                ……..
    }
 ……..
}
```

Состав модулей и структур данных ТКП представлен на рисунке 15.

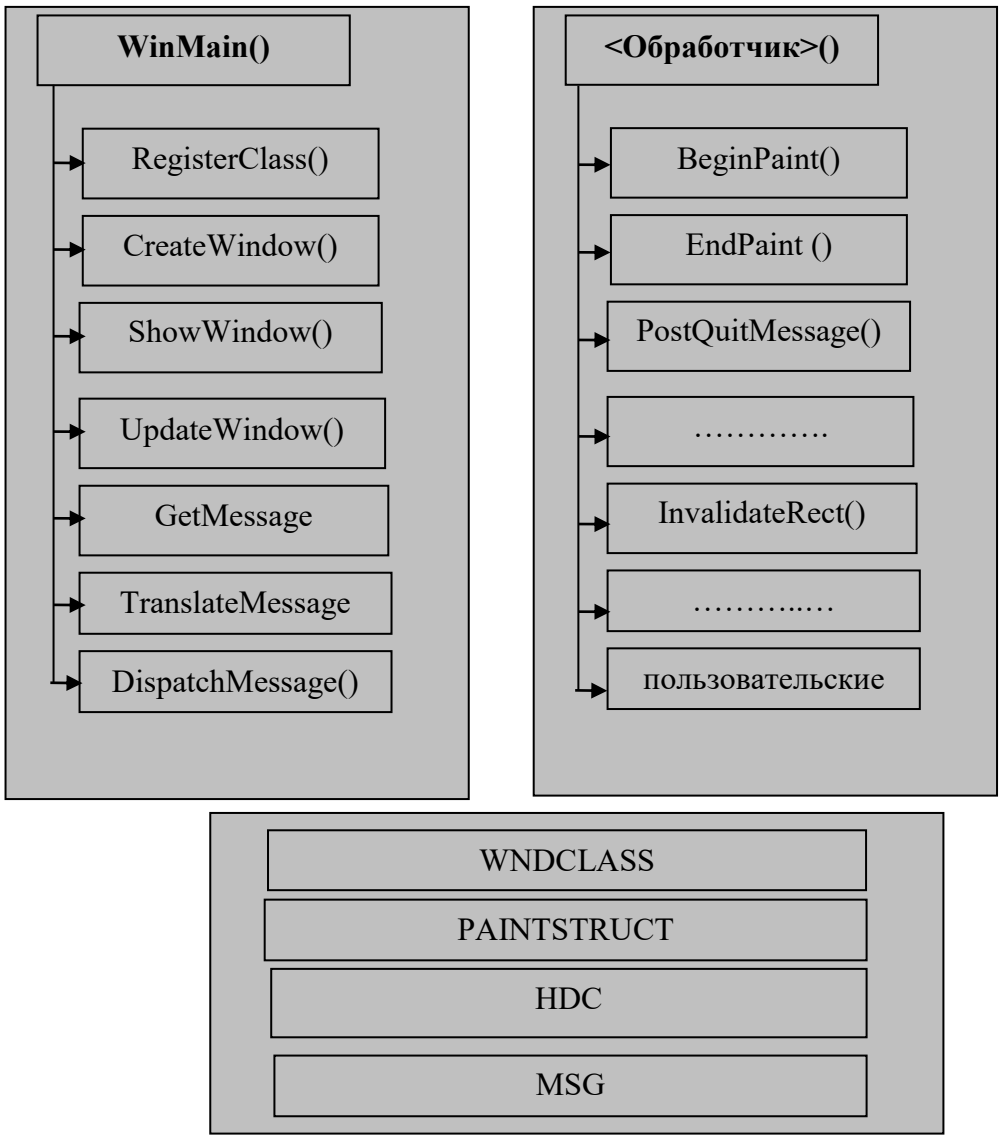


Рисунок 15 - Функциональный состав ТКП

2 Схема технологии разработки приложений на базе ТКП

Приложение при проектировании может рассматриваться как совокупность однотипных компонентов, включающих окна и их графические ресурсы (описания окон и их элементов) и программное обеспечение в виде функций-обработчиков (описания функций, реализующих реакцию на сообщения и выполняющих необходимую предметную обработку).

РАЗРАБОТКА приложения включает следующие этапы.

1. Проектирование.

1.1. Проектирование интерфейсного компонента.

1.2. Проектирование программного компонента.

2. Реализация.

2.1. Реализация интерфейсных компонентов.

2.1.1. Создание файла описания ресурсов.

2.1.2. Кодирование интерфейсных компонентов.

2.1.3. Подключение к приложению файла описания ресурсов.

2.2. Реализация (кодие) проваиирограммных компонентов. Инициализация и визуализация интерфейсных компонентов.

Этапы разработки приложений на базе ТКП. 1. Проектирование. Результат – спецификация компонентов приложения.

1.1. Проектирование интерфейсного компонента. Включает проектирование окна и его элементов. Результат - вид (облик), состав элементов (кнопок, меню, и т.д.), их свойства (атрибуты), события, сообщения, спецификация реакций на сообщения.

Для окна (главного окна) проектируется пользовательское меню, определяются ресурсы окна (кисти, перья, пиктограммы и т.п.).

Для диалогового окна проектируются элементы управления (кнопки, поля редактирования и т.п.).

1.2. Проектирование программного компонента (программного обеспечения). Результат – алгоритмы работы функций-обработчиков по спецификациям реакций на сообщения.

2. Реализация. Результат – исполнимый код приложения.

2.1. Реализация интерфейсных компонентов. Означает описание на соответствующих языках интерфейсных компонентов приложения (графических ресурсов).

2.1.1. Создание файла описания ресурсов.

2.1.2. Кодирование интерфейсных компонентов. Означает описание на соответствующих языках интерфейсных компонентов приложения (графических ресурсов).

Для окна (главного окна) создается стиль окна с названием <ИмяСтиляОкна>. В дальнейшем зарегистрированный стиль окна с названием <ИмяСтиляОкна> может использоваться для создания и визуализации конкретного окна (окон). А после его создания для управления окном в функциях используется его < ДескрипторОкна >. Для создания стиля окна:

- кодируются ресурсы, при необходимости - пользовательское меню с идентификатором <ИмяРесурсаМеню>. Ресурсы кодируются с помощью специальных команд описания или визуально в редакторе ресурсов. Фиксируются <ID_идентификаторы> ресурсов, например, исполнимых пунктов меню (кнопок, команд);

- определяется имя функции - <ИмяОбработчикаОкна>;

- описывается стиль окна с названием <ИмяСтиляОкна>, путем задания значений атрибутов окна в полях структуры данных описания стиля окна типа WNDCLASS, где в том числе указываются <ДескрипторПриложения> (как параметр функции WinMain), <Имя-

СтиляОкна>, <ИмяРесурсаМеню>, <ИмяОбработчикаОкна>, <Пиктограмма>, <КистьФона> и т.п.;

- стиль окна связывается с его обработчиком путем указания его названия <ИмяОбработчикаОкна> в соответствующем поле структуры данных описания окна WNDCLASS;

- стиль окна регистрируется в ОС функцией RegisterClass (tagWNDCLASS *), в которую передается ссылка на заполненную структуру данных описания стиля окна типа WNDCLASS.

Для диалогового окна описываются составляющие его ресурсы с названием <ID_РесурсовДиалоговогоОкна>. В дальнейшем название созданных <ID_РесурсовДиалоговогоОкна> может использоваться для создания и визуализации конкретного окна (окон). А после его создания для управления окном в функциях используется его < ДескрипторОкна >. Для создания ресурсов диалогового окна:

- выполняется описание его элементов управления с помощью команд описания или визуально в редакторе ресурсов;

- в редакторе ресурсов определяется название диалогового окна <ID_РесурсовДиалоговогоОкна> и <ID_идентификаторы> его элементов управления (кнопок, списков и т.п.).

2.1.3. Подключение к приложению файла описания ресурсов.

2.2. Реализация (кодирование) программных компонентов. Означает описание на соответствующих языках программных компонентов приложения - функций-обработчиков. В том числе этот этап предполагает выполнение таких действий как инициализация и визуализация интерфейсных компонентов. Означает задание значений дополнительных атрибутов, связывание с функциями-обработчиками и как результат – получение < ДескрипторaОкна >, вывод окна на экран.

Для окна (главного окна) связывание с функциями-обработчиками было выполнено при описании стиля окна. Поэтому здесь только уточняются параметры инициализации - <ЗаголовокОкна>, < КоординатаЛевогоВерхнегоУглаОкна >, < ШиринаОкна >, < ВысотаОкна >, стиль окна. Выполняется указанное с помощью функции

HWND <ДескрипторОкна> = CreateWindow (LPSTR <ИмяСтиляОкна>, …).

Окно визуализируется функцией

ShowWindow (<ДескрипторОкна>, <ИзображениеОкнаПриПервомВыводе>).

Для диалогового окна инициализация, включая связывание ресурсов (<ID_РесурсовДиалоговогоОкна>) с приложением hInstance (<ДескрипторПриложения>), родительским окном hWnd (<ДескрипторРодительскогоОкна >) и обработчиком сообщений диалогового окна (<ИмяОбработчикаОкна>), а также визуализация производится одной функцией (аналогом функций CreateWindow и ShowWindow)

DialogBox (<ДескрипторПриложения>, <ДескрипторДиалоговогоОкна>,
<ДескрипторРодительскогоОкна >, <ИмяОбработчикаОкна>); .

Схема описания, реализации главного окна ТКП Окно является основным компонентом интерфейса приложения. С понятием окно связаны графические ресурсы окна (пиктограмма, кисть, меню и т.п.) и функция-обработчик сообщений окна. Создание и работа с окнами базируется на понятии стиль (класс) окна. Основные этапы представлены ниже.

1. Пользователь по своему усмотрению определяет переменную для обозначения стиля, класса окна, на который можно ссылаться при его инициализации

char szWindowClass [] = <ИмяСтиляОкна> ; .

2. Атрибуты стиля задаются в структуре данных WNDCLASS для хранения информации о стиле окна. После регистрации стиля окна в ОС <ИмяСтиляОкна> может использоваться для создания конкретных экземпляров окна. В структуре данных WNDCLASS, в частности, задается параметр, доопределяющий класс, стиль окна (например, параметр CS_VREDRAW обеспечивается перерисовку содержимого клиентской области окна при изменении размера окна по вертикали).

Для этого описывается переменная

WNDCLASS <СтруктураДанныхОкна>

и задаются атрибуты стиля, <СтруктураДанныхОкна>.lpszClassName = (LPCSTR) <ИмяСтиляОкна> и т.д.

3. Окно регистрируется в ОС. Регистрация окна производится функцией RegisterClass, в которую передается структура данных типа WNDCLASS, описывающая стиль окна

RegisterClass (tagWNDCLASS * <СтруктураДанныхОкна>).

4. Экземпляр окна создается функцией

```
HWND  <ДескрипторОкна>  = CreateWindow (LPSTR <ИмяСтиляОкна>,
                    LPSTR  <ЗаголовокОкна>,
                    DWORD <ДополнительныйСтиль2> = WS_OVERLAPPEDWINDOW,
                    int < КоординатаЛевогоУглаОкнаX > = CW_USEDEFAULT,
                    Int < КоординатаЛевогоУглаОкнаУ > = CW_USEDEFAULT,
                    Int < ШиринаОкна > = CW_USEDEFAULT,
                    Int < ВысотаОкна > = CW_USEDEFAULT,
                    HWND <РодительскоеОкно > = (HWND)NULL,
                    HMENU <Меню> = (HMENU) NULL,
                    HANDLE <ДескрипторПриложения> = hInstance,
                    LPSTR <ДопИнформация> = NULL);  .
```

Здесь <ДополнительныйСтиль2> – доопределяет класс, стиль окна (например, параметр WS_OVERLAPPEDWINDOW обеспечивает создание перекрывающегося окна с системным типовым меню, типовыми кнопкам). См. ПРИЛОЖЕНИЕ Д. Значения параметра dwStyle функции CreateWindow.

Теперь <ДескрипторОкна> можно использовать в функциях управления окном.

5. Визуализация окна производится функцией ShowWindow с прототипом

ShowWindow (HWND <ДескрипторОкна>, int <ИзображениеОкнаПриПервомВыводе>)

Здесь параметр <ИзображенияОкнаПриПервомВыводе>, задает способ его вывода. Значение либо берется из соответствующего параметра - int nCmdShow, переданного ОС в функции WinMain, или переустанавливается пользователем по его усмотрению. см. ПРИЛОЖЕНИЕ Е. Значения параметра nCmdShow функции ShowWindow (доопределение состояния окна при начальном запуске)

Общая схема функционирования приложений на базе ТКП иллюстрируется рисунком 16.

3 ТКП. Создание в системе Visual Studio

Ниже описана структура Windows-приложения на базе ТКП с типовым минимальным набором графических ресурсов.

Соответственно интерфейс такого приложения базируется на использовании классического окна с клиентской областью (в качестве главного) с пользовательским стилем (зарегистрированным здесь как – myWindowStyle). Это стиль (окно), определяющий:

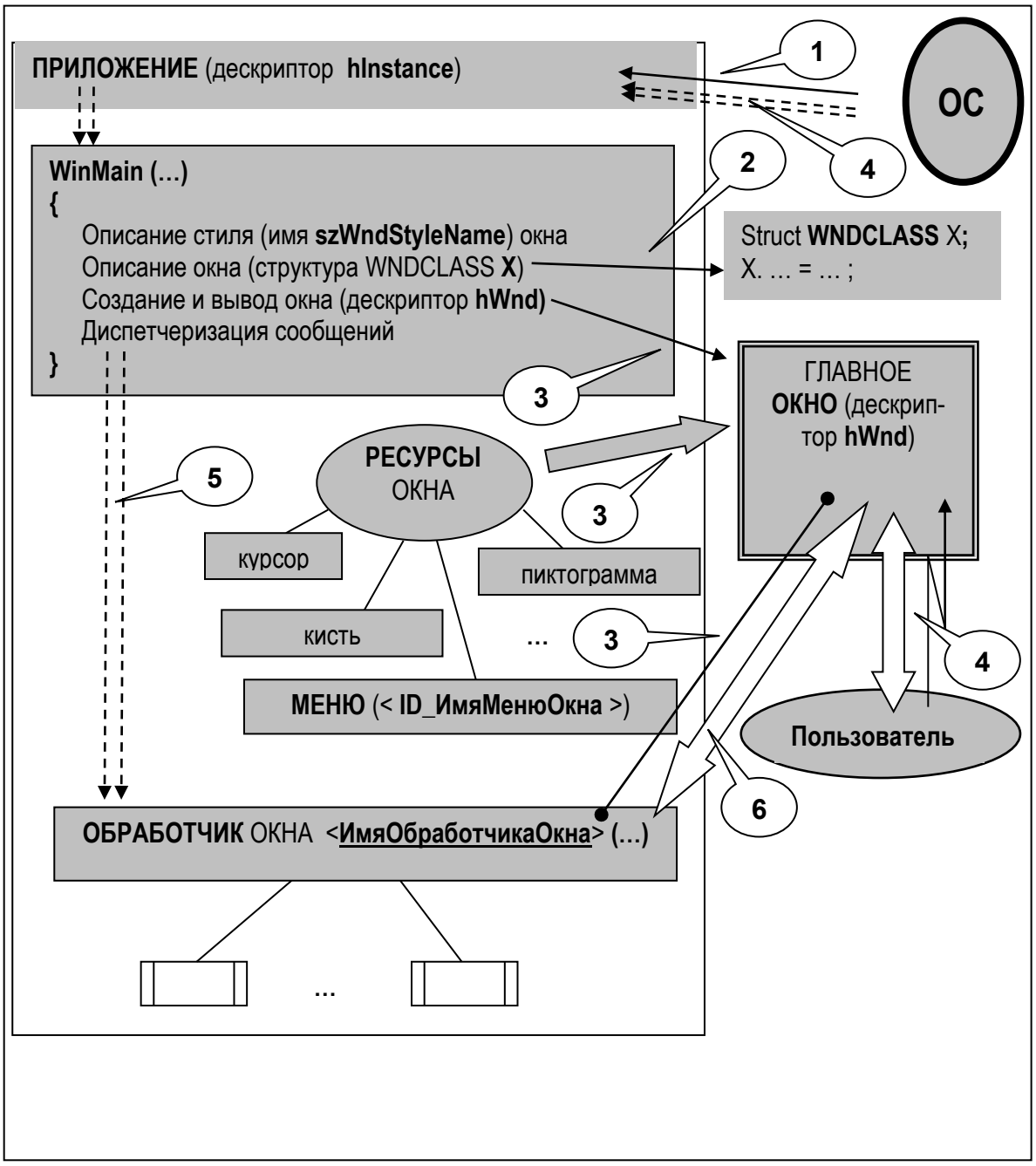


Рисунок 16 - Схема обработки сообщений в ТКП

- отсутствие пользовательского меню; - перерисовку содержимого окна при изменении его размеров (параметры CS_HREDRAW, CS_VREDRAW; - светлый фон (параметр WHITE_BRUSH); - курсор типа “стрелка” (параметр IDC_ARROW). С окном связан обработчик сообщений (здесь с именем WndProc).

При создании окна (в функции CreateWindow) дополнительно указаны:

- заголовок ("ТИПОВОЙ КАРКАС Windows-приложения … "); - стиль - перекрывающееся окно с системным меню и системными кнопками (параметр WS_OVERLAPPEDWINDOW).

При визуализации окна в параметрах функции ShowWindow (hWnd, nCmdShow) указан вывод окна в развернутом виде. Значения параметра nCmdShow функции ShowWindow используется для доопределения состояния окна при начальном запуске. Соответственно окно выглядит как показано ниже (рисунок 17).

Рисунок 17 - Интерфейс ТКП

Пример текста приложения приведен ниже.

```
#include <windows.h>        // описания Windows
LRESULT CALLBACK       WndProc (HWND, UINT, WPARAM, LPARAM);
char   szWindowStyle [ ] = "myWindowStyle";

//==============================
//   ГЛАВНАЯ ФУНКЦИЯ
int  WINAPI  WinMain   (HINSTANCE hInst,  //дескриптор приложения
                        HINSTANCE hPreInst,  //всегда NULL
                        LPSTR        lpszCmdLine,  //командная строка
                        int             nCmdShow) //окно при первом выводе
{
     HWND                    hWnd;              //дескриптор окна
     MSG                     lpMsg;             //структура хранения сообщений
     WNDCLASS                wcApp;             //структура описания стиля окна
     wcApp.lpszClassName   = szWindowStyle;   //имя стиля окна
     wcApp.hInstance       = hInst;           //дескриптор приложения
     wcApp.lpfnWndProc     = WndProc;         //указатель на обработчик сообщений
     wcApp.hCursor         = LoadCursor(NULL, IDC_ARROW);   //курсор - “стрелка”
     wcApp.hIcon           = 0;                //без использования пиктограммы
     wcApp.lpszMenuName    = 0;                //дескриптор меню (без меню)
     wcApp.hbrBackground   = (HBRUSH) GetStockObject (WHITE_BRUSH); //цвет фона
     wcApp.style           = CS_HREDRAW | CS_VREDRAW; //перерисовывать окно
     wcApp.cbClsExtra      = 0;                //число доп. байт для WNDCLASS
     wcApp.cbWndExtra      = 0;                //общее число доп. байт

     if    ( ! RegisterClass (&wcApp) )  //регистрация окна
             return  0;

    hWnd = CreateWindow (szWindowStyle,
```

```
                    "ТИПОВОЙ КАРКАС Windows-приложения … ",
                     WS_OVERLAPPEDWINDOW, //окно перекрывающееся, меню, кнопки
                       CW_USEDEFAULT,       //координата x - левый верхний угол окна
                       CW_USEDEFAULT,       //координата y - левый верхний угол окна
                       CW_USEDEFAULT,       //ширина окна в единицах устройства
                       CW_USEDEFAULT,       //высота окна в единицах устройства
                       (HWND) NULL,         //указатель на родительское окно
                       (HMENU) NULL,        //зависит от стиля окна (указатель меню)
                        hInst,              //дескриптор приложения
                        NULL );      //адрес дополнительной информации об окне
      ShowWindow (hWnd, nCmdShow);      //вывод окна
      UpdateWindow (hWnd);              //перерисовка окна
       while (GetMessage ( &lpMsg, NULL, 0, 0) )
       {
            TranslateMessage (&lpMsg);       //преобразование виртуальных клавиш
             DispatchMessage (&lpMsg);       //передача сообщения обработчику
       }
       return ( lpMsg.wParam );
}

//==============================
//   ФУНКЦИЯ-ОБРАБОТЧИК ГЛАВНОГО ОКНА. (имя выбирает пользователь)
LRESULT CALLBACK  WndProc (HWND  hWnd,          //дескриптор окна
                           UINT    messg,       //код сообщения
                           WPARAM  wParam,  LPARAM    lParam)
{
    HDC          hdc;          //дескриптор контекста устройства
    PAINTSTRUCT ps;            //структура для клиентской области окна
    switch (messg)
    {
        case WM_PAINT:                 //перерисовать окно
            hdc = BeginPaint (hWnd, &ps);
            //-----Начало фрагмента пользователя
            //-----Конец фрагмента пользователя
            ValidateRect (hWnd,NULL);
            EndPaint (hWnd, &ps);
            break;
        case WM_DESTROY:     //послать сообщение о завершении приложения
             PostQuitMessage (0);
             break;
        default:
             return ( DefWindowProc (hWnd, messg, wParam, lParam));
      }
      return 0;
}
```

Аналогичный по функциональности и интерфейсу текст можно получить автоматически, используя мастер построения каркасов системы Visual Studio - Win32 Application (Simple) или Application (Hello). Однако в последнем случае он будет отличаться от приведенного здесь структурированием – составом модулей, функций.

Ниже представлен код ТКП с указанием фрагментов, доопределяемых пользователем

```
//========== описания глобальных объектов =============
LRESULT CALLBACK  <ИмяОбработчикаОкна> (HWND, UINT, WPARAM, LPARAM);
char   szWindowStyle [ ] = <ИмяСтиляОкна>;

//================ГЛАВНАЯ ФУНКЦИЯ==================
int  WINAPI  WinMain (HINSTANCE hInst,  HINSTANCE hPreInst,
                      LPSTR  lpszCmdLine,    int   nCmdShow)
{
     HWND                    hWnd;
     MSG                     lpMsg;
     WNDCLASS                wcApp;
     wcApp.lpszClassName   = szWindowStyle;
     wcApp.hInstance       = hInst;
     wcApp.lpfnWndProc     = <ИмяОбработчикаОкна>;
     wcApp.hCursor         = LoadCursor (NULL, IDC_ARROW);
     wcApp.hIcon           = 0;
     wcApp.lpszMenuName    = 0;
     wcApp.hbrBackground   = (HBRUSH) GetStockObject (WHITE_BRUSH);
     wcApp.style           = CS_HREDRAW | CS_VREDRAW;
     wcApp.cbClsExtra      = 0;
     wcApp.cbWndExtra      = 0;
     if    ( ! RegisterClass (&wcApp))  return  0;
     hWnd = CreateWindow  (szWindowStyle, "КАРКАС ПРИЛОЖЕНИЯ",
                           WS_OVERLAPPEDWINDOW, CW_USEDEFAULT,
                           CW_USEDEFAULT, CW_USEDEFAULT,
                           CW_USEDEFAULT, (HWND) NULL,
                           ( HMENU )NULL,  hInst,  NULL);
     ShowWindow (hWnd, nCmdShow);
     UpdateWindow (hWnd);
     while ( GetMessage (&lpMsg, NULL, 0, 0) )
     {
          TranslateMessage (&lpMsg);
          DispatchMessage  (&lpMsg);
     }
     return (lpMsg.wParam);
}

//=== ФУНКЦИЯ-ОБРАБОТЧИК СООБЩЕНИЙ ГЛАВНОГО ОКНА  ===
LRESULT CALLBACK  <ИмяОбработчикаОкна>(HWND hWnd, UINT  Mess,
                                       WPARAM  wParam, LPARAM lParam)
{
     HDC              hdc;
     PAINTSTRUCT      ps;
     switch (Mess)
     {
          case WM_PAINT:
               hdc = BeginPaint (hWnd, &ps);
               //< ФРАГМЕНТ ПОЛЬЗОВАТЕЛЯ >
```

```
                ValidateRect (hWnd,NULL);
                EndPaint (hWnd, &ps);
               break;
          //case  < КодСообщения > :
                //hdc = BeginPaint (hWnd, &ps);
                //< ФРАГМЕНТ ПОЛЬЗОВАТЕЛЯ >
                //ValidateRect (hWnd,NULL);
                //EndPaint (hWnd, &ps);
                //break;
           case WM_DESTROY:
                PostQuitMessage (0);
                break;
           default:
                return DefWindowProc (hWnd, Mess, wParam, lParam);
     }
     return  0;
}
```

Лекция № 6 Последовательность работы с окнами

1 Создание класса (стиля) окна. Структура WNDCLASS

При создании приложения создается два основных компонента: - окно, как интерфейсное средство общения пользователя с приложением; - функция, обрабатывающая все события (сообщения), происходящие как в окне так и во внешней среде приложения. Окно описывается разработчиком. Данные о каждом окне хранятся в полях структуры WNDCLASS (в функции WinMain это переменная типа WNDCLASS wcApp). Переменная для обозначения стиля, класса окна, на который можно ссылаться при его инициализации, задается пользователем

```
char   szWindowClass [ ] = <ИмяСтиляОкна> ;  .
```

Например,

```
char   szWindowClass [ ] = "myWindowClass"; .
```

Пример инициализации структуры WNDCLASS приведен ниже, значения полей структуры указаны ниже (таблицы 8-11).

Таблица 8 - Поля структуры WNDCLASS

Имя поля	Значение
Style	определяет стиль класса. Различные стили могут комбинироваться при помощи побитовой операции ("логическое ИЛИ")
lpfnWndProc	указатель на функцию-обработчик окна
cbClsExtra	число дополнительных байтов, выделяемых структуре WNDCLASS (обычно NULL
cbClsExtra	число дополнительных байтов, выделяемых для всех дополнительных структур окна (обычно NULL)
hInstance	дескриптор экземпляра приложения, регистрирующий стиль окна
hIcon	пиктограмма окна в свернутом виде (обычно NULL)

hCursor	тип курсора, используемый для данного окна (обычно NULL)
hbrBackGround	тип кисти, используемой для фона окна (значением данного параметра может быть как идентификатор физической кисти, так и значение цвета)
lpszMenuName	указатель строки – имени, дескриптора ресурса - меню (по умолчанию NULL)
lpszClassName	указатель строки – уникального имени класса окна

Таблица 9 - Некоторые стандартные курсоры

Имя курсора	Форма курсора
IDC_ARROW	стандартный указатель типа "стрелка"
IDC_CROSS	указатель типа “перекрестие”
IDC_IBEAM	текстовый курсор
IDC_WAIT	указатель типа “песочные часы”

Таблица 10 - Некоторые стандартные кисти

Имя кисти, цвет кисти	Тип фона
BLACK_BRUSH	черный
DKGRAY_BRUSH	темно-серый
HOLLOW_BRUSH	прозрачный
LTGRAY_BRUSH	светло-серый
WHITE BRUSH	белый
COLOR_SCROLLBAR, COLOR_BACKGROUND, COLOR_HIGHLIGHT, COLOR_ACTIVECAPTION, COLOR_HIGHLIGHTTEXT, COLOR_WINDOW, COLOR_WINDOWTEXT, COLOR_CAPTIONTEXT, COLOR_APPWORKSPACE	при использовании цвета к его значению нужно прибавить 1

Таблица 11 - Значения параметра Style

Параметр	Значение
CS_BYTEALIGNCLIENT (CS_BYTEALIGNWINDOW)	использовать границы по байту по оси X и производить выравнивание клиентской области окна (использовать границы по байту по оси X и производить выравнивание окна)
CS_DBCLKS	посылать окну сообщения о двойном щелчке "мыши"
CS_HREDRAW	перерисовывать содержимое клиентской области окна при изменении размера окна по горизонтали
CS_KEYCVTWINDOW	выполнять преобразование виртуальных клавиш
CS_NOCLOSE	без пункта закрытия окна в системном меню
CS_NOKEYCVT	не выполнять преобразование виртуальных клавиш
CS_OWNDC	присваивать каждому экземпляру окна свой контекст устройства
CS_PARENTDC	окну передавать контекст устройства родительского окна
CS_SAVEBITS	сохранять часть окна, перекрытую на экране другим окном
CS_VREDRAW	перерисовывать окно при изменении размера по вертикали

При этом <ИмяСтиляОкна> может использоваться для создания конкретных экземпляров окна

Например,

```
WNDCLASS      wcApp;
wcApp.lpszClassName   = myWindowClass;
wcApp.hInstance  = hInst;
wcApp.lpfnWndProc   = WndProc;
wcApp.hCursor   = LoadCursor (NULL, IDC_ARROW);  // курсор окна - стрелка
wcApp.hIcon   =  0;    //без пиктограммы при выводе окна в свернутом виде
wcApp.lpszMenuName   = 0;  // без меню
wcApp.hbrBackground   = (HBRUSH) GetStockObject (WHITE_BRUSH);
wcApp.style = CS_HREDRAW|CS_VREDRAW;// перерисовывать окно при изменении размеров
wcApp.cbClsExtra        = 0;
wcApp.cbWndExtra      = 0;      .
```

Пример использования расширенной структуры WNDCLASSEX приведен ниже

```
WNDCLASSEX   wcex;
wcex.cbSize = sizeof (WNDCLASSEX);
wcex.style     = CS_HREDRAW | CS_VREDRAW;
wcex.lpfnWndProc   = (WNDPROC) WndProc;
wcex.cbClsExtra      = 0;
wcex.cbWndExtra    = 0;
wcex.hInstance       = hInstance;
wcex.hIcon           = LoadIcon ( hInstance, (LPCTSTR) IDI_WORK);
wcex.hCursor         = LoadCursor (NULL, IDC_ARROW);
wcex.hbrBackground          = (HBRUSH) (COLOR_WINDOW+1);
wcex.lpszMenuName           = (LPCSTR) IDC_ WORK;
wcex.lpszClassName          = szWindowClass;
wcex.hIconSm                = LoadIcon(wcex.hInstance, (LPCTSTR) IDI_SMALL);  .
```

Регистрация окна производится функцией RegisterClass, в которую передается структура данных типа WNDCLASS, описывающая стиль окна. Прототип функции

Unsigned short__cdecl RegisterClass (tagWNDCLASS * <СтруктураДанныхОкна>)

Например

```
if  ( !RegisterClass (&wcApp) )
            return 0;  .
```

2 Создание и визуализация окна

Когда стиль (класс, вид) окна описан и зарегистрирован в Windows под соответствующим именем (здесь имя задано строкой – szWindowClass), то может быть создано произвольное число таких окон и получены их дескрипторы для дальнейшего использования. Это выполняется с помощью функции CreateWindow.

Эта функция конкретизирует вид создаваемого окна в соответствии со своими параметрами, а ОС Windows возвращает его уникальный номер, т.е. дескриптор (handle) – переменную типа HWND hWnd = CreateWindow (…). Параметры CreateWindow, доопределяющие стиль окна, приведены в таблице ЧЧЧ. Сама визуализация окна выполняется функцией ShowWindow (hWnd, nCmdShow). Значения второго параметра функции (доопределение состояния окна при начальном запуске) приведены в таблицах 12, 13.

Таблица 12 - Значения параметра dwStyle

Параметр стиля	Описание
WS_BORDER	создание окна с рамкой
WS_CAPTION	добавление к окну с рамкой заголовка
WS_CHILD	создание дочернего окна
WS_CHILDWINDOW	создание дочернего окна стиля WS_CHILD
WS_CLIPCHILDREN	при создании родительского окна запрещает его рисование в области, занятой любым дочерним окном
WS_CLIPSIBLINGS	(только со стилем WS_CHILD) не использовать при получении данным окном сообщения о перерисовке области, занятые другими дочерними окнами. Иначе разрешается рисовать в клиентской области другого дочернего окна
WS_DISABLED	создание первоначально неактивного окна
WS_DLGFRAME	создание окна с двойной рамкой без заголовка
WS_EX_ ACCEPTFILES	создание окна, поддерживающего метод ""drag-and-drop"
WS_EX_ DLGMODALFRAME	(только для значения dwExStyle) создание окна с двойной рамкой и дополнительным заголовком
WS_EX_ NOPARENTNOTIFY	создание дочернего окна, которое не будет посылать родительскому окну сообщение WM_PARENTNOTIFY при создании или разрушении
WS_EX_TOPMOST	созданное окно должно располагаться поверх всех окон, не имеющих этого стиля, и оставаться поверх остальных даже в неактивном состоянии
WS_EX_ TRANSPARENT	созданное окно должно быть прозрачным, т.е. все окна, расположенные под данным окном, не заслоняются им
WS_GROUP	(только в диалоговых окнах) стиль указывается для первого элемента управления в группе элементов, пользователь перемещается между ними с помощью клавиш-стрелок
WS_HSCROLL	создание окна с горизонтальной полосой прокрутки
WS_MAXIMIZE WS_MINIMIZE	создание окна максимального (минимального) размера
WS_MAXIMIZEBOX WS_MINIMIZEBOX	создание окна с кнопкой максимизации (минимизации)
WS_OVERLAPPED	создание перекрывающегося окна
WS_OVERLAPPED WINDOW	создание перекрывающегося окна на базе стилей WS_OVERLAPPED, WS_THICKFRAME и WS_SYSMENU
WS_POPUP	(не используется с окнами стиля WS_CHILD) создание временного окна
WS_	создание временного окна на базе стилей WS_BORDER,

POPUPWINDOW	WS_POPUP и WS_SYSMENU
WS_SYSMENU	(только для окон с заголовком) создание окна с кнопкой вызова системного меню вместо стандартной кнопки, позволяющей закрыть окно при использовании этого стиля для дочернего окна
WS_TABSTOP	(только в диалоговых окнах) указывает на произвольное количество элементов управления (стиль WS_TABSTOP), между которыми можно перемещаться клавишей <TaЬ>
WS_THICKFRAME	создание окна с толстой рамкой, используемой для изменения размеров окна
WS_VISIBLE	создание окна, видимого сразу после создания
WS_VSCROLL	создание окна с вертикальной полосой прокрутки

Таблица 13 - Состояния окна при запуске (параметры в ShowWindow)

Параметр состояния	Описание
SW_HIDE	прячет окно и переводит в активное состояние другое окно
SW_MIINIMIZE	минимизирует окно и активизирует окно верхнего уровня
SW_RESTORE	действует так же, как и SW_SHOWNORMAL
SW_SHOW	активизирует окно и выводит его в текущей позиции и текущего размера
SW_SHOWDEFAULT	активизирует окно и выводит его с использованием текущих умолчаний
SW_SHOWMAXIMIZED	активизирует окно и выводит его с максимальным размером
SW_ SHOWMINIMIZED	активизирует окно и выводит его в виде пиктограммы
SW_SHOWMINNOACTIVATE	выводит окно как пиктограмму, при этом ранее активное окно сохраняет свою активность
SW_SHOWNA	выводит окно с учетом его состояния в данный момент; (активное в данный момент окно остается активным)
SW_SHOWNOACTIVATE	выводит окно в его прежней позиции и прежнего размера (активное в данный момент окно остается активным)
SW_SHOWNORMAL	активизирует окно и выводит его на экран. Если окно было увеличено или уменьшено до пиктограммы, то система Windows восстановит начальное положение и размер окна
SW_SHOWSMOOTH	Выводит окно так, чтобы оно минимально перекрывалось другими окнами

Прототип функции создания окна приложения CreateWindow приведен ниже

```
HWND CreateWindow ( LPSTR    lpszClassName,
                    LPSTR    lpszWindowName,
                    DWORD    dwStyle,
                    int      x,
                    int      y,
                    int      Width,
                    int      Height,
                    HWND     hWndParent,
```

HMENU hMenu,
HANDLE hInst,
LPSTR lpParam); .

```
HWND <ДескрипторОкна> =
CreateWindow (LPSTR <ИмяСтиляОкна>,
              LPSTR <ЗаголовокОкна>,
              DWORD <ДополнительныйСтиль2> = WS_OVERLAPPEDWINDOW,
              int < КоординатаЛевогоУглаОкнаX > = CW_USEDEFAULT,
              Int < КоординатаЛевогоУглаОкнаУ > = CW_USEDEFAULT,
              Int < ШиринаОкна > = CW_USEDEFAULT,
              Int < ВысотаОкна > = CW_USEDEFAULT,
              HWND <РодительскоеОкно > = (HWND)NULL,
              HMENU <Меню> = (HMENU) NULL,
              HANDLE <ДескрипторПриложения> = hInstance,
              LPSTR <ДопИнформация> = NULL); .
```

Здесь

- HWND <ДескрипторОкна> = CreateWindow (…) – дескриптор окна, используемый в функциях управления окном, например ShowWindow;
- LPSTR lpszClassName = <ИмяСтиляОкна> - имя зарегистрированного стиля, класса окна;
- LPSTR lpszWindowName - заголовок (название) окна;
- DWORD dwStyle <ДополнительныйСтиль2> - доопределяет класс, стиль окна (например, параметр WS_OVERLAPPEDWINDOW обеспечивает создание перекрывающегося окна с системным типовым меню, типовыми кнопкам);
- int x – задает начальное положение окна по оси X;
- int y – задает начальное положение окна по оси Y;
- int Width - задает ширину окна в единицах устройства;
- int Height - задает высоту окна в единицах устройства;
- HWND hWndParent - указатель на родительское окно (у перекрывающегося окна его нет);
- HMENU hMenu - указывает на меню окна;
- HANDLE hInst = <ДескрипторПриложения> - определяет копию модуля, связанного с окном (модуля, который создал окно);
- LPSTR lpParam - адрес дополнительной информации, нужной для создания окна.

Например,

```
hWnd = CreateWindow( szWindowClass, “ПРИМЕР”, WS_OVERLAPPEDWINDOW,
          CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL); .
```

Визуализация окна производится функцией ShowWindow с прототипом

int__cdecl ShowWindow (HWND hWnd, int nCmdShow) .

```
int__cdecl ShowWindow (HWND <ДескрипторОкна>,
                       int <ИзображениеОкнаПриПервомВыводе>)
```

Здесь
- HWND hWnd - <ДескрипторОкна>, которое надо вывести на экран;
- int nCmdShow - способ <ИзображенияОкнаПриПервомВыводе>, чье значение либо берется из соответствующего параметра, переданного ОС в функции WinMain, или переустанавливается пользователем по его усмотрению.

Для диалоговых окон аналогом функций CreateWindow и ShowWindow служит функция DialogBox. Она инициализирует диалоговое окно, связывая его ресурсы (<ID_РесурсовДиалоговогоОкна>) с приложением hInstance (<ДескрипторПриложения>), родительским окном hWnd (<ДескрипторРодительскогоОкна >) и обработчиком сообщений диалогового окна (<ИмяОбработчикаОкна>). Прототип функции

```
DialogBox  (HANDLE hInstance,
            LPCSTR lpszDialogName,
            HWND hWnd,
            DLGPROC lpfnDlgProc);
```

DialogBox (HANDLE **<ДескрипторПриложения> ,**
LPCSTR **<ДескрипторРесурсовДиалоговогоОкна>**
= (LPCSTR) **<ID_РесурсовДиалоговогоОкна>**,
HWND <**ДескрипторРодительскогоОкна** >,
DLGPROC **<ИмяОбработчикаОкна>)**;

Здесь
- HANDLE hInstance - <ДескрипторПриложения>;
- LPCSTR lpszDialogName - <ID_РесурсовДиалоговогоОкна>;
- HWND hWnd - <ДескрипторОкна >;
- DLGPROC lpfnDlgProc - <ИмяОбработчикаОкна>.

Например,

```
DialogBox( hInst, (LPCTSTR) IDD_DIALOG1, hWnd,
           (DLGPROC)  TO_PROCESS_DIALOG_BOX_MESSAGES);
DialogBox( hInst, (LPCTSTR) IDD_ABOUTBOX, hWnd, (DLGPROC) About);  .
```

Управление окном приложения. Обновление (перерисовка) окна производится функцией UpdateWindow, т.к. она генерирует сообщение WM_PAINT

```
int__cdecl UpdateWindow (HWND hWnd);
```

получение контекста устройства для окна производится функцией

```
HDC BeginPaint (hWnd, &ps);
```

освобождение контекста устройства

EndPaint (hWnd, &ps);

обновление клиентской области окна (посылка сообщения на перерисовку) вызывается функцией

ValidateRect (hWnd, NULL) .

Завершение работы с окном приложения. Работа с окном приложения завершается функцией (при этом посылается сообщение WM_DESTROY)
int__cdecl DestroyWindow (HWND hWnd); .

Если закрытие окна должно привести и к завершению работы с приложением, то можно обработать сообщение WM_DESTROY с помощью функции (при этом посылается сообщение WM_QUIT)
VOID __cdecl PostQuitMessage (int);

Закрытие диалогового окна производится функцией
int__cdecl EndDialog (HWND hWnd, int <КодЗавершения>); .

Лекция № 7 Графические ресурсы

1 Редакторы ресурсов. Создание ресурсов

В Windows-приложениях для хранения описаний ресурсов используется ресурсный файл типа ResourceScript, который должен быть создан и подключен к приложению командой #include. А сами ресурсы могут создаваться “вручную” (например, путем описания меню в текстовом редакторе), либо с помощью редакторов ресурсов. Последние используются для визуального проектирования ресурсов. Это редакторы меню, диалоговых окон, инструменты для работы со значками, растровыми изображениями и т.п. Доступ к встроенным редакторам ресурсов осуществляется из пункта главного меню Resource.

1. Для создания нового файла ресурсов следует, открыв главное окно среды разработки Visual Studio, выполнить команду добавления в готовый проект соответствующего файла описания ресурсов: пункт меню Project, подпункт Add to Project, New, вкладка Files, тип файла ResourceScript.

2. Для добавления нового ресурса с использованием соответствующего редактора ресурсов следует, открыв главное окно среды разработки Visual Studio, выбрать пункт меню Insert, подпункт Resource. На экран будет выведено окно с перечнем доступных ресурсов (рисунок 18). При выборе типа ресурсов автоматически будет вызван соответствующий редактор ресурсов.

Рисунок 18 - Типы ресурсов

Это ресурсы следующих типов.

Акселератор (Accelerator) для настройки комбинаций "горячих" клавиш.

Битовый образ (Bitmap) - небольшой по размеру цветной графический объект в виде растрового описания, отображающий окно, часть окна, объекты типа “стрелка”, “кисть”, “курсор” и т.п. используемый для быстрого вывода соответствующего изображения на экран. Вызываемый при этом редактор графических изображений позволяет создавать и модифицировать растровые изображения пользовательских курсоров, пиктограмм, сохраняемых в файле с расширением *.rc и включаемых в файлы сценариев ресурсов.

Курсор (Cursor), в том числе текстовый для отображения позиции ввода, курсор - указатель “мыши” (см. битовый образ).

Пиктограмма (Icon) – графический объект (см. битовый образ).

Диалоговые окна (Dialog) для описания соответствующих окон и расположенных на них элементах управления. Вызываемый при этом редактор диалоговых окон — это средство разработки графических объектов, позволяющее быстро создавать сложные диалоговые окна с возможностью комбинировать, изменять и настраивать в соответствии с собственными требованиями элементы окна, элементы управления окна. Элементы имеют набор заранее определенных свойств, а их настройка сводится к изменению значений этих свойств.

Текст в формате HTML (HTML).

Меню (Menu) - для создания иерархических пользовательских меню.

Таблица (String Table) - для хранения выводимой текстовой информации. Вызываемый при этом редактор строк предназначен для обработки таблиц строк, представляющих собой ресурс, содержащий список идентификаторов (заголовков), используемых в приложении (одно приложение - одна таблица). Например, здесь могут храниться сообщения, отображаемые в строке состояния. Таблица упрощает изменение языка интерфейса программы, т.к. достаточно перевести на другой язык строки таблицы, не затрагивая код программы.

Панель инструментов (Toolbar).

Информация о версии проекта (Version).

Сами окна (например, с рамкой), включающие клиентскую область, имеют такие собственные ресурсы как перо, кисть, шрифт. Перо – невидимый до рисования гра-

фический объект, применяемый для изображения линий, контуров. Основные атрибуты пера – стиль линии, ее ширина, цвет. Кисть – невидимый до рисования графический объект, применяемый для закраски, заполнения областей изображения. Основные атрибуты кисти – размер, стиль рисования (наполнитель), цвет. Шрифт – графический объект, описываемый как набор символов одного семейства с точки зрения начертания (т.е. одного размера, стиля). Основные атрибуты – толщина, размер, тип начертания (курсив, с подчеркиванием и т.д.).

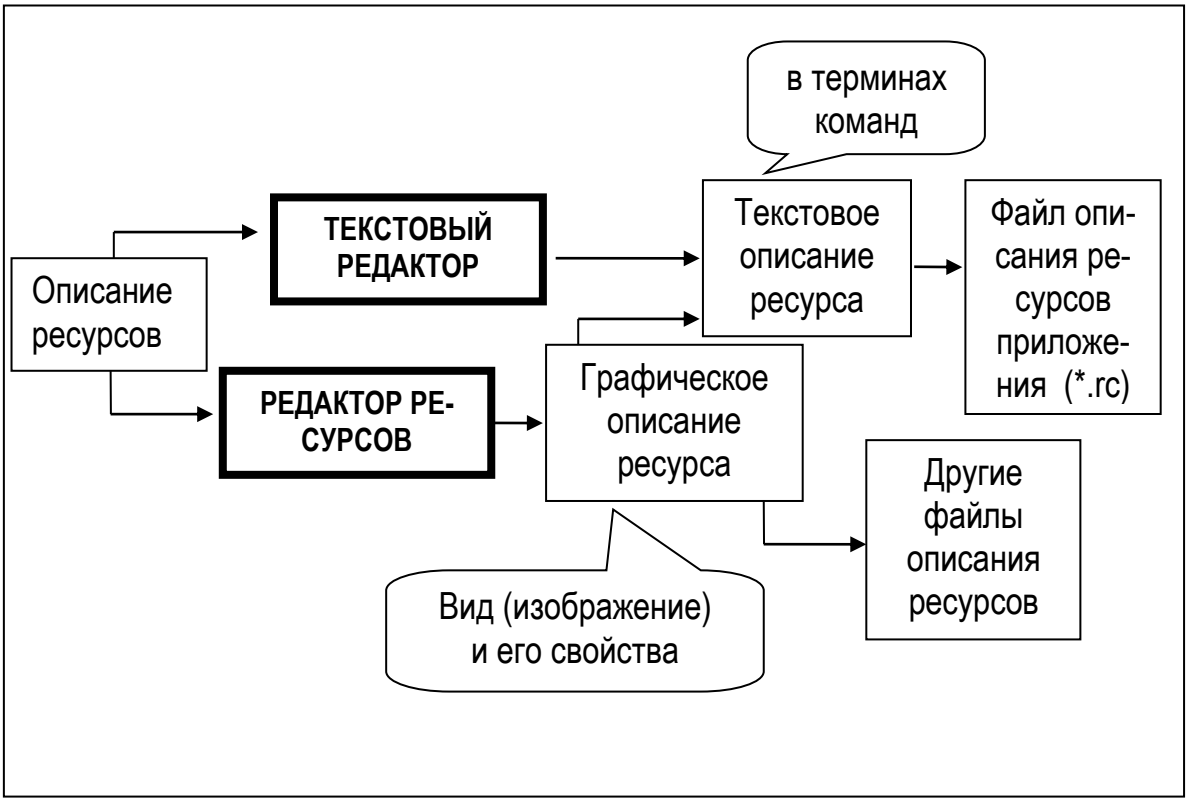


Рисунок 19 - Обработка ресурсов

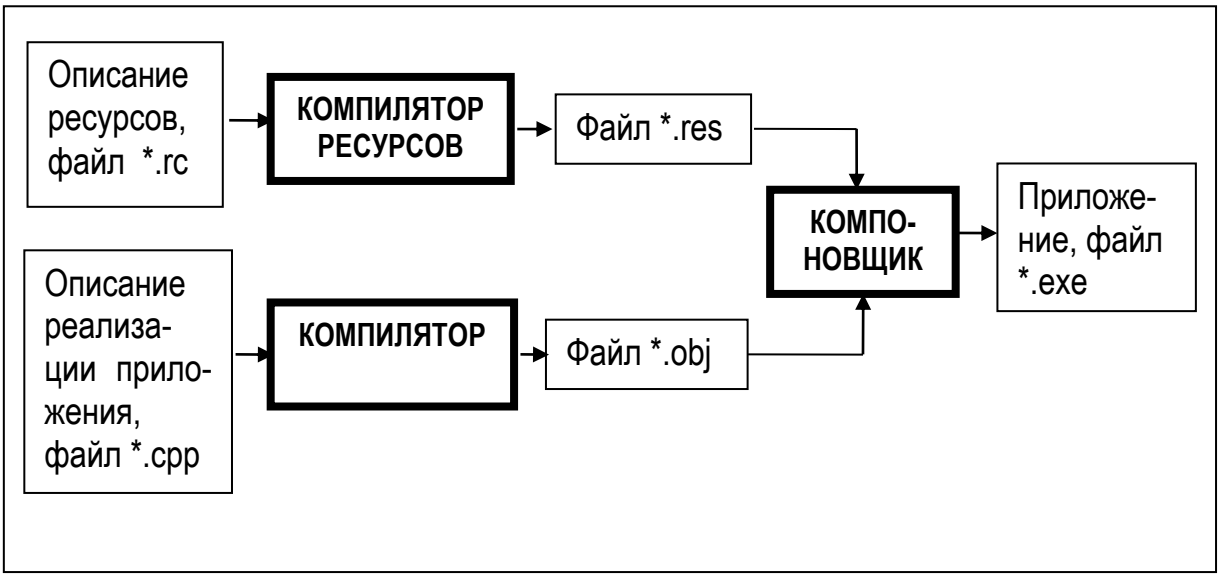


Рисунок 20 - Компоновка приложения

3. После того как ресурс создан в редакторе ресурсов или “вручную”, компилятор ресурсов считывает ASCII-файл описания ресурсов (*.rc) и создает для компоновщика приложения его двоичный аналог - res-файл.

Порядок обработки ресурсов и компоновки приложения иллюстрируется на рисунках 19, 20, а расширения используемых файлов приведены в таблице 14.

Таблица 14 - Типы файлов, поддерживаемых средой разработки

rc	файл сценария (script)
ret	файл шаблона (template)
res	файл ресурсов
exe	исполняемый файл (редактируется только в операционных системах, построенных на платформе Windows NT. В операционных системах Windows 9x/ME может быть только открыт, но не изменен)
dll	файл динамически подключаемой библиотеки (по возможностям редактирования аналогичен файлу с расширением exe)
bmp	графический файл (“Windows”- формат)
ico, cur	графический файл для хранения изображения пиктограммы

2 Диалоговое окно в составе главного окна

Интерфейс приложения часто строится на базе масштабируемого окна с рамкой, с клиентской областью (особенно, если предполагается использование клиентской области для вывода текстовой, графической информации), которое может содержать главное меню и обеспечивать запуск других окон, включая диалоговые.

Для разработки приложения с диалоговым окном необходимо спроектировать и создать ресурс - диалоговое окно: - разработать вид и состав окна (например, окно с названием, системной кнопкой закрытия и двумя пользовательскими кнопками с именами OK и Cancel); - определить функциональный состав приложения, состав и алгоритмы соответствующих обработчиков.

Параметры стилей диалоговых окон приведены в таблице 15.

Описания обработчиков проиллюстрированы ниже.

Первый вариант листинга приведен на примере каркаса типа Hello для поддерживаемого им справочно-информационного окна About с одинаковой реакцией на нажатие кнопки OK и нажатие кнопки CANCEL – выход из справки.

```
LRESULT CALLBACK About(HWND hDlg, UINT message,
                                WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
    case WM_INITDIALOG: return TRUE;
    case WM_COMMAND:
        if ( LOWORD(wParam ) == IDOK ||
           LOWORD(wParam ) == IDCANCEL )
        {
            EndDialog( hDlg, LOWORD(wParam) );
            return TRUE;
        }
```

```
        break;
        }
        return FALSE;
}
```

Второй вариант листинга приведен на примере диалогового окна с двумя кнопками OK и CANCEL с раздельными секциями для обработки сообщений – нажатие кнопки OK и нажатие кнопки CANCEL.

```
LRESULT CALLBACK TO_PROCESS_MESSAGE (HWND hDlg, UINT message,
                                                    WPARAM wParam, LPARAM lParam)
{
        switch (message)
        {
        case WM_INITDIALOG:
                // < ФРАГМЕНТ_ПОЛЬЗОВАТЕЛЯ >
                return FALSE;
        case WM_COMMAND:
                switch (wParam)
                {
                case IDOK:
                        // < ФРАГМЕНТ_ПОЛЬЗОВАТЕЛЯ >
                        EndDialog(hDlg,TRUE);
                        break;
                case IDCANCEL:
                        // < ФРАГМЕНТ_ПОЛЬЗОВАТЕЛЯ >
                        EndDialog(hDlg,FALSE);
                        break;
                default:  return FALSE;
                }
                break;
        default:  return FALSE;
        }
        return TRUE;
}; .
```

Для инициализации окна и вывода его на экран надо связать приложение (*ДескрипторПриложения* - переменная hInstance), родительское окно (*ДескрипторОкна*), диалоговое окно (*ДескрипторРесурса* – например, идентификатор IDD_DIALOG1), обработчик сообщений диалогового окна (*ИмяОбработчика*).

Все это выполняется функцией с прототипом DialogBox(HINSTANCE *ДескрипторПриложения*, LPCTSTR *ДескрипторРесурса*, HWND *ДескрипторОкна*, DLGPROC *ИмяОбработчика*). Запуск указанной функции приведет к визуализации и активизации диалогового окна.

ПРИМЕР. *Создать Windows-приложение, при запуске которого в качестве главного должно выводиться масштабируемое окно с рамкой и клиентской областью, а за ним сразу же автоматически диалоговое окно с кнопками. При этом, главное окно остается на экране, но теряет активность. Нажатие кнопок диалогового окна приводит к его закрытию. Примерный вид оконного интерфейса приведен ниже* (рисунок 21).

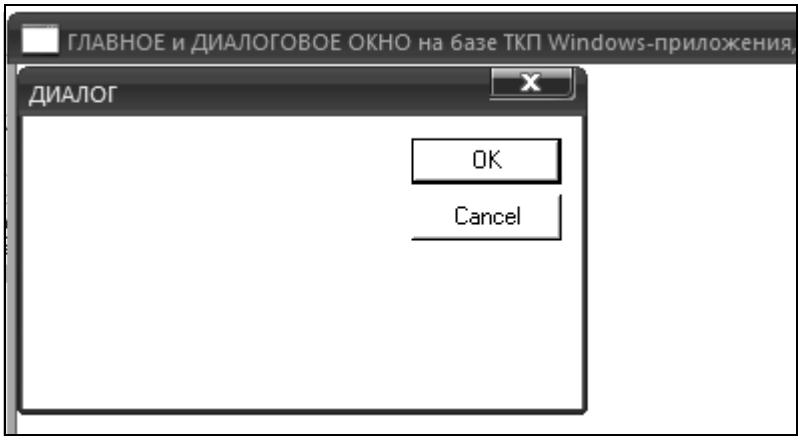


Рисунок 21 - Оконный интерфейс

Таблица 15 - Наиболее распространенные стили диалоговых окон

Значение	Действие
DS_MODALFRAME	создается диалоговое окно с модальной рамкой (этот стиль может использоваться как для модальных, так и для немодальных окон)
WS_BORDER	создается окно с обычной рамкой
WS_CAPTION	создается окно со строкой заголовка
WS_CHILD	окно создается как дочернее
WS_HSCROLL	создается окно с горизонтальной полосой прокрутки
WS_MAXIMIZEBOX	создается окно с кнопкой максимизации
WS_MINIMIZEBOX	создается окно с кнопкой минимизации
WS_POPUP	окно отображается поверх всех других окон
WS_SYSMENU	создается окно с системным меню
WS_TABSTOP	элементы управления окна могут быть выбраны последовательным нажатием клавиши TAB
WS_VISIBLE	окно отображается при вызове
WS_VSCROLL	создается окно с вертикальной полосой прокрутки

Ниже описана функция диалогового окна - обработчик его сообщений с прототипом:

LRESULT CALLBACK *ИмяОбработчика* (HWND, UINT, WPARAM, LPARAM);

```
LRESULT CALLBACK ИмяОбработчика ( HWND    hDlg, UINT   Message,
                                  WPARAM wParam,  LPARAM lParam)
{
    switch (Message)
    {
     case WM_INITDIALOG:
            return FALSE;
      case WM_COMMAND:
            switch (wParam)
            {
            case IDOK:
                  EndDialog(hDlg,TRUE);
                   break;
```

```
            case IDCANCEL:
                    EndDialog(hDlg,FALSE);
                     break;
             default:
                     return FALSE;
            }
            break;
         default:
            return FALSE;
        }
        return TRUE;
   }; .
```

Визуализация и активизация диалогового окна выполняется, например, как DialogBox (hInstance, (LPCTSTR) IDD_DIALOG1, hWnd, (DLGPROC) TO_PROCESS_DIALOG_BOX)).

3 Диалоговое окно в качестве главного окна

Диалоговое окно может использоваться в интерфейсе приложения в роли главного окна. Например, при управлении приложением с помощью набора простых команд, составляющих один уровень иерархии (как “добавить запись”, “найти запись”, “удалить запись” и т.п.), которые могут быть представлены в окне набором соответствующих кнопок.

ПРИМЕР. *Создать Windows-приложение, при запуске которого в качестве главного должно выводиться диалоговое окно с кнопками. Нажатие кнопок диалогового окна приводит к подтверждению получения соответствующего сообщения, а нажатие Cancel - к его закрытию. Примерный вид оконного интерфейса приведен ниже.*

Назначение приложения. Приложение содержит одно окно – диалоговое, которое в качестве главного окна выводится при его запуске. Демонстрируется реакция приложения на нажатия кнопок окна ОК и CANCEL и нажатие левой кнопки “мыши”.

Интерфейсные формы. Включают диалоговое окно в роли главного с двумя командными кнопками ОК и CANCEL. При нажатии кнопки ОК или нажатии левой кнопки “мыши” выводятся окна сообщений (рисунки 22, 23). При нажатии кнопки CANCEL приложение завершается.

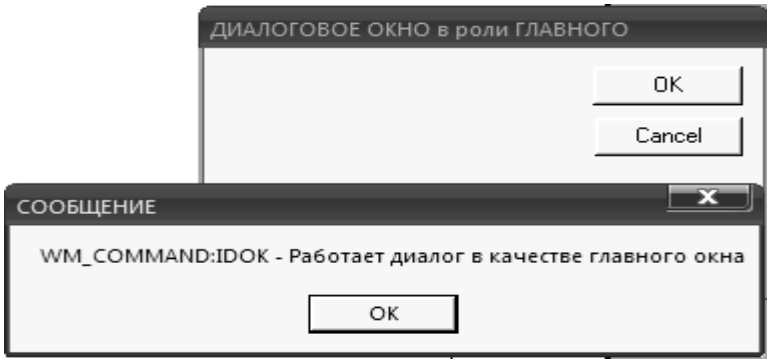


Рисунок 22 - Фрагмент интерфейса

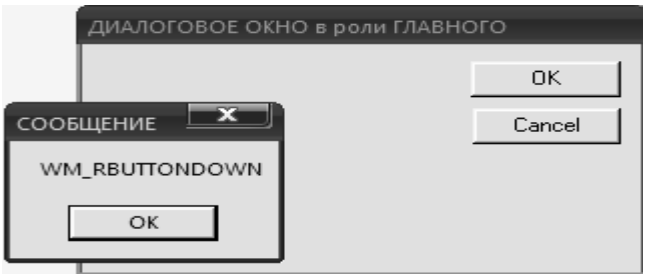


Рисунок 23 - Фрагмент интерфейса

Обрабатываемые сообщения. Это сообщения, адресуемые главному окну: - сообщение WM_RBUTTONDOWN; - сообщение WM_COMMAND от ресурса IDOK; - сообщение WM_COMMAND от ресурса IDCANCEL.

По нажатии кнопки ОК инициируется сообщение WM_COMMAND: IDOK, что приводит к выводу информационного сообщения о приложении (в виде MessageBox).

По нажатии кнопки CANCEL инициируется сообщение WM_COMMAND: IDCANCEL, что при его обработке приводит к разрушению диалогового окна (функцией EndDialog(hDlg,TRUE)) и закрытию приложения путем посылки ОС сообщения PostQuitMessage(0) на закрытие приложения).

По нажатии левой кнопки “мыши” в пределах диалогового окна приложения инициируется сообщение WM_RBUTTONDOWN, что приводит к выводу информационного сообщения о событии (в виде MessageBox).

В главной функции WinMain вместо действий по описанию, регистрации, созданию и визуализации классического главного окна (с рамкой) инициализировать и визуализировать диалоговое окно. Для этого связать текущее приложение (например, HINSTANCE hInst), ранее созданный ресурс - диалоговое окно (например, IDD_DIALOG1) и его обработчик командой DialogBox, например

```
DialogBox(hInst, (LPCTSTR) IDD_DIALOG1, NULL, (DLGPROC) ИмяОбработчика ) ;
```

и описать обработчик

```
LRESULT CALLBACK ИмяОбработчика ( HWND    hDlg, UINT   Message,
                                  WPARAM wParam,  LPARAM lParam)
{
     switch (Message)
     {
      case WM_INITDIALOG:
            return FALSE;
      case WM_RBUTTONDOWN:
            < ФрагментПользователя >
            break;
       case WM_COMMAND:
            switch (wParam)
            {
            case IDOK:
                  < ФрагментПользователя >
                   break;
            case IDCANCEL:
                  < ФрагментПользователя >
```

```
                break;
            default:
                return FALSE;
            }
            break;
        default:
            return FALSE;
        }
        return TRUE;
};   .
```

Диалоговое окно с окном редактирования как главное окно. При использовании диалогового окна его пространство используется для размещения различных элементов управления, обеспечивающих, в том числе ввод-вывод данных и управление работой.

ПРИМЕР. *Создать Windows-приложение, при запуске которого в качестве главного должно выводиться диалоговое окно (с окнами редактирования, подсказками, кнопками), предназначенное для задания исходных данных и их эхо-вывода. По кнопке Ввести производится ввод значения из окна ввода и эхо-вывод введенного значения.*

Интерфейсные формы. Примерный вид диалогового окна показан ниже (рисунок 24). Окно включает два окошка редактирования (ЭУ типа Edit Box), два статичных окна (ЭУ типа Static Text), командную кнопку Ввести.

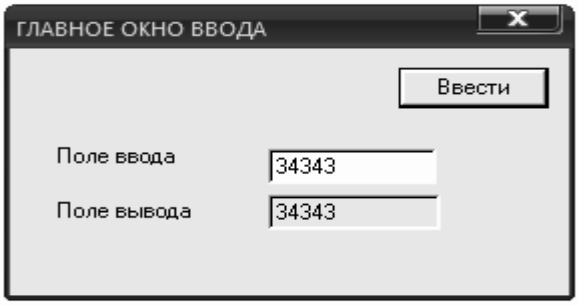


Рисунок 24 - Фрагмент интерфейса

Обрабатываемые сообщения. Это сообщения, адресуемые диалоговому окну: - сообщение по нажатии кнопки Ввести (например, WM_COMMAND: ID_Enter). Действие – считывание содержимого окна редактирования (например, IDC_EDIT1) в строку (например, InputString) с последующим выводом строки в окно редактирования (например, IDC_EDIT2). Для этого в обработчике окна (в секции case ID_Enter) следует вставить команды GetDlgItemText, SetDlgItemText; - сообщение case WM_COMMAND:IDCANCEL по нажатии системной кнопки завершения работы. Действие – разрушение диалогового окна функцией EndDialog и закрытие приложения путем посылки ОС сообщения PostQuitMessage.

Состав ресурса диалоговое окно. Окошки редактирования (например, с IDC_EDIT1, IDC_EDIT2 для ввода и вывода соответственно), статические окна (например, с IDC_STATIC1, IDC_STATIC2 соответственно с названиями Caption - “Поле ввода”, “Поле вывода”), кнопка (например, с ID_Enter).

Требования к обработчику. В секции case ID_Enter описать строковую переменную char InputString[256], ввести данные из окна ввода в строку; вывести данные из строки в

окно вывода. В секции IDCANCEL выполнить функции по завершению работы с приложением EndDialog(hDlg,TRUE), PostQuitMessage(0).

Диалоговое окно со списком и окном редактирования. Теоретические сведения. Список отображает набор строк. Строку можно выбрать “мышью”, клавиатурой. При этом строка выделяется цветом и посылается сообщение от списка системе о событии. Информацию об элементе управления список - List Box можно найти в MSDN (в разделе List Box Controls, местонахождение Technical Articles). Там описаны следующие компоненты списка.

Стили (Styles) списка имеют названия LBS_<НазваниеСтиля>. Например, LBS_STANDARD, LBS_SORT (строки списка располагаются в алфавитном порядке), LBS_MULTIPLESEL (с возможностью выбора произвольного числа строк), LBS_MULTICOLUMN (многоколонковый список с горизонтальной прокруткой), LBS_NOTIFY (с отсылкой родительскому окну сообщения о выборе строки) (рисунок 25).

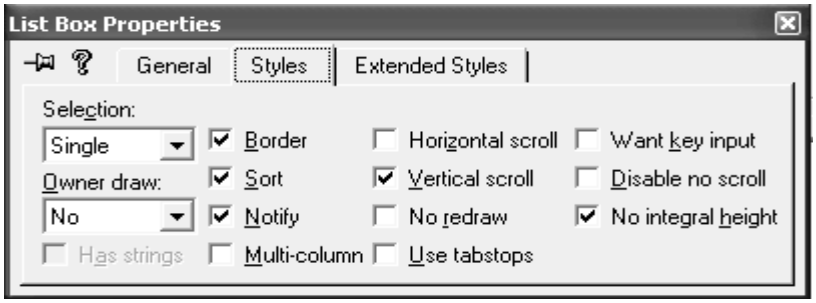


Рисунок 25 - Настройка списка

Сообщения (Notification Messages), адресуемые своему родительскому окну от списка о происходящих с ним событиях, именуются как LBN_<НазваниеСобытия>. Например, LBN_DBLCLK, LBN_SELCHANGE. Родительское окно списка принимает эти сообщения в виде соответствующих WM_COMMAND сообщений.

Сообщения – команды (Messages), адресуемые списку от приложения именуются как LB_<НазваниеКоманды>. Например, LB_ADDSTRING, LB_DELETESTRING, LB_SETCURSEL, LB_GETCURSEL, LB_GETTEXT, LB_GETCOUNT.

Для посылки сообщения (*КодСообщения* с параметрами *ПараметрыСообщения1*, *ПараметрыСообщения2*) указанному ЭУ (*ДескрипторЭУ*) указанного диалогового окна (ДескрипторДиалоговогоОкна) в приложениях используется функция SendDlgItemMessage (для списка *ДескрипторЭУ = ДескрипторСписка,* КодСообщения = *НазваниеКоманды*, а значения *ПараметровСообщения* зависят от типа команды)

SendDlgItemMessage (HWND *ДескрипторДиалоговогоОкна*, int *ДескрипторЭУ*, UINT *КодСообщения*, WPARAM *ПараметрыСообщения1*, LPARAM *ПараметрыСообщения2*) .

Примеры использования команды приведены ниже:

- получение текущего размера списка выполняется командой

ЧислоСтрокСписка = SendDlgItemMessage(<ДескрипторОкна>, <ДескрипторСписка>, LB_GETCOUNT, 0, 0) ,

int RecordsAmount = SendDlgItemMessage(hDlg, IDC_LIST1, LB_GETCOUNT, 0, 0) .

- добавление строки выполняется командой

SendDlgItemMessage (<ДескрипторОкна>,<ДескрипторСписка>,LB_ADDSTRING,
0, (LPARAM) <ДобавляемаяСтрока>) ,

SendDlgItemMessage(hDlg, IDC_LIST1, LB_ADDSTRING, 0, (LPARAM) "Иванов") .

- удаление строки выполняется командой

SendDlgItemMessage (<ДескрипторОкна>, <ДескрипторСписка>, LB_DELETESTRING, <НомерУдаляемойСтроки>, 0) ,

SendDlgItemMessage(hDlg, IDC_LIST1, LB_DELETESTRING, i, 0).

- установка текущей позиции выделения выполняется командой

SendDlgItemMessage (<ДескрипторОкна>, <ДескрипторСписка>, LB_SETCURSEL, 0, (LPARAM) <ВыделяемаяСтрока>)

SendDlgItemMessage(hDlg, IDC_LIST1, LB_SETCURSEL, 0, (LPARAM) "Иванов");

- получение номера текущей позиции выделения выполняется командой

НомерВыделеннойСтроки = SendDlgItemMessage(<ДескрипторОкна>, <Дескриптор-Списка>, LB_GETCURSEL, 0, 0) ,

i = SendDlgItemMessage(hDlg, IDC_LIST1, LB_GETCURSEL, 0, 0);

- получение текста строки по указанному номеру выполняется командой

SendDlgItemMessage (<ДескрипторОкна>, <ДескрипторСписка>, LB_GETTEXT, <НомерСтроки>, (LPARAM) <ПриемнаяСтрока>)

SendDlgItemMessage(hDlg, IDC_LIST1, LB_GETTEXT, i, (LPARAM) Text) .

ПРИМЕР. *Разработать приложение с диалоговым окном в роли главного*.

Цель приложения. Управление списком строк – фамилий (просмотр, выбор, добавление, удаление, редактирование, расположение в алфавитном порядке). При запуске приложения в качестве главного окна выводится диалоговое окно, содержащее пустой список, средства управления списком и кнопку Завершить (Cancel).

Интерфейсные формы. Примерный вид диалогового окна представлен ниже (рисунок 26). Соответственно состав ресурсов: - список (например, с IDC_LIST_NAMES), окно редактирования (например, с IDC_EDIT_NAME), кнопки Добавить, Удалить, Изменить, Завершить (например, с IDC_BUTTON_ADD, IDC_BUTTON_DELETE, IDC_BUTTON_EDIT, IDCANCEL), рамки – ЭУ типа Group Box и т.д.

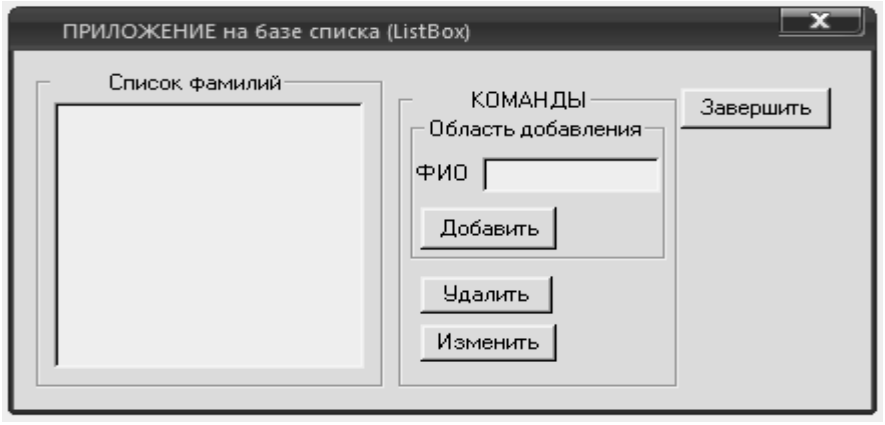


Рисунок 26 - Интерфейс приложения

Здесь обрабатываются следующие группы сообщений.

Сообщения к окну (целиком):

– WM_COMMAND: IDCANCEL - по нажатии кнопки ЗАВЕРШИТЬ приложение функцией EndDialog(hDlg,TRUE) закрывает диалог и функцией PostQuitMessage(0) извещает систему о необходимости завершить приложение. Для этого генерируется и адресуется окну сообщение UINT message = WM_COMMAND с параметром int wmId = LOWORD(wParam) = IDCANCEL. Оно обрабатывается в обработчике в секции case WM_COMMAND: case IDCANCEL:.

- WM_INITDIALOG - при запуске приложения и визуализации окна командой DialogBox возможна подготовка окна к работе, его инициализация (например, начальное заполнение списка константной информацией или считывание информации из файла в список). Для этого генерируется и адресуется окну сообщение UINT message = WM_INITDIALOG. Оно обрабатывается в обработчике в секции case WM_INITDIALOG:.

Сообщения, относящиеся к ЭУ (списку) окна:

– WM_COMMAND: IDC_BUTTON_ADD - по нажатии кнопки ДОБАВИТЬ считывается функцией GetDlgItemText (hDlg, IDC_EDIT_NAME, Строка, ДлинаСтроки) фамилия, набранная в поле IDC_EDIT_NAME. Если это не пустая строка (функция strcmp), то (через функцию SendDlgItemMessage) приложение генерирует и адресует списку (IDC_<ДескрипторСписка>) сообщение LB_ADDSTRING – добавить к списку указанную в функции строку, считанную ранее из окна редактирования фамилию. Для этого генерируется и адресуется окну сообщение UINT message = WM_COMMAND с параметром int wmId = LOWORD(wParam) = IDC_BUTTON_ADD. Оно обрабатывается в обработчике в секции case WM_COMMAND: case IDC_BUTTON_ADD:.

– (WM_COMMAND: IDC_BUTTON_DELETE - по нажатии кнопки УДАЛИТЬ приложение считывает (через функцию SendDlgItemMessage) номер выбранной строки списка (как результат обработки адресуемого списку сообщения LB_GETCURSEL). Далее приложение (через функцию SendDlgItemMessage) генерирует и адресует списку (IDC_<ДескрипторСписка>) сообщение LB_DELETESTRING – удалить из списка строку с указанным в функции номером, считанным ранее. Для этого генерируется и адресуется окну сообщение UINT message = WM_COMMAND с параметром int wmId = LOWORD(wParam) = IDC_BUTTON_DELETE. Оно обрабатывается в обработчике в секции case WM_COMMAND: case case IDC_BUTTON_DELETE:.

– WM_COMMAND: IDC_BUTTON_EDIT - по нажатии кнопки ИЗМЕНИТЬ приложение считывает (через функцию SendDlgItemMessage) номер выбранной строки списка (как результат обработки адресуемого списку сообщения LB_GETCURSEL). Далее приложение (через функцию SendDlgItemMessage) генерирует и адресует списку (IDC_<ДескрипторСписка>) сообщение LB_GETTEXT – получить из списка строку с указанным в функции номером, считанным ранее. Считанная строка помещается в поле IDC_EDIT_NAME (функция SetDlgItemText) для редактирования с одновременным стиранием этой строки из списка (через функцию SendDlgItemMessage с параметром LB_DELETESTRING) (последующее добавление отредактированной строки к списку инициируется кнопкой ДОБАВИТЬ).

Для обработки рассматриваемого события генерируется и адресуется окну сообщение UINT message = WM_COMMAND с параметром int wmId = LOWORD(wParam) = IDC_BUTTON_EDIT. Оно обрабатывается в обработчике в секции case WM_COMMAND: case IDC_BUTTON_EDIT:.

Сообщения, относящиеся к окну (целиком) от ЭУ (списка) окна:

– WM_COMMAND: IDC_LIST_NAMES: LBN_DBLCLK - при выборе строки в списке фамилий, например, двойным щелчком, генерируется сообщение LBN_DBLCLK. Здесь действие приложения сводится просто к выводу подтверждающего сообщения о происшедшем событии. Для этого генерируется и адресуется окну сообщение UINT message = WM_COMMAND с параметром int wmId = LOWORD(wParam) = IDC_LIST_NAMES с дополнительным параметром int wmEvent = HIWORD(wParam) = LBN_DBLCLK. Оно обрабатывается в обработчике в секции case WM_COMMAND: case IDC_LIST_NAMES: case LBN_DBLCLK:.

– WM_COMMAND: IDC_LIST_NAMES: LBN_SELCHANGE - при выборе строки в списке фамилий, например, щелчком, клавиатурой генерируется сообщение об изменении выделения LBN_SELCHANGE. Здесь действие приложения сводится просто к выводу под-

тверждающего сообщения о происшедшем событии. Для этого генерируется и адресуется окну сообщение UINT message = WM_COMMAND с параметром int wmId = LOWORD(wParam) = IDC_LIST_NAMES с дополнительным параметром int wmEvent = HIWORD(wParam) = LBN_SELCHANGE. Оно обрабатывается в обработчике в секции case WM_COMMAND: case IDC_LIST_NAMES: case LBN_SELCHANGE.

Ниже представлена общая структура обработчика

```
LRESULT CALLBACK DlgProc(…)
{
    int  wmId    = LOWORD(wParam);
    int  wmEvent = HIWORD(wParam);
    int  RecordsAmount =
                SendDlgItemMessage(hDlg, IDC_LIST_NAMES, LB_GETCOUNT, 0, 0);
    switch (Message)
    {
    case WM_INITDIALOG:  … break;
    case WM_COMMAND:
        switch (wmId)
        {
        case IDCANCEL: … break;
        case IDC_BUTTON_ADD: … break;
        case IDC_BUTTON_DELETE: … break;
        case IDC_BUTTON_EDIT: … break;
        case IDC_LIST_NAMES:
            switch (wmEvent)
            {
            case LBN_SELCHANGE: … break;
            case LBN_DBLCLK: … break;
            default: …
            }
            break;
        default: …
        }
        break;
    default:
    }
}   .
```

Вариант кода обработчика сообщений приложения (с кнопками OK и CANCEL, элементами управления типа Edit, List) приведен ниже

```
LRESULT CALLBACK DlgProc (HWND hDlg, UINT message,
                                    WPARAM wParam,  LPARAM lParam)
{
  int i; int j =0;
  int wmId, wmEvent;
  wmId      = LOWORD (wParam);
  wmEvent = HIWORD (wParam);
  int  RecordsAmount = SendDlgItemMessage ( hDlg, IDC_LIST1, LB_GETCOUNT, 0, 0 );
  char SelectedText[34];
```

```
    switch (message)
    {
    case WM_INITDIALOG:
        SendDlgItemMessage ( hDlg, IDC_LIST1, LB_ADDSTRING, 0,
                             ( LPARAM) "Иванов  233345" );
        SendDlgItemMessage ( hDlg, IDC_LIST1, LB_ADDSTRING, 0,
                             ( LPARAM) "Сидоров  673345" );
        SendDlgItemMessage ( hDlg, IDC_LIST1, LB_SETCURSEL, 0,
                             (LPARAM) "Иванов  233345" );
        break;
    case WM_COMMAND:
        switch (wmId)
        {
        case IDCANCEL:
                EndDialog (hDlg,TRUE);
                PostQuitMessage (0);
                break;
        case IDC_BUTTON_Edit:
                i = SendDlgItemMessage (hDlg, IDC_LIST1, LB_GETCURSEL, 0, 0);
                if ( ( I >= 0 ) && ( I < RecordsAmount ) )
                {
                        SendDlgItemMessage( hDlg, IDC_LIST1, LB_GETTEXT, i,
                                                ( LPARAM) SelectedText );
                        char Str1[78], Str2[78];
                        sscanf ( SelectedText,"%s %s", TheRecord.Name, Str2 );
                        SetDlgItemText ( hDlg, IDC_EDIT_Name, TheRecord.Name );
                        SetDlgItemText ( hDlg, IDC_EDIT_Number, Str2 );
                        SendDlgItemMessage ( hDlg, IDC_LIST1, LB_DELETESTRING, i, 0 );
                }
                break;
        case IDC_LIST1:
                switch (wmEvent)
                {
                case LBN_DBLCLK:
                        //Обработка двойного щелчка по элементу списка
                        break;
                }
                break;
                default:   return FALSE;
        }
        break;
        default:   return FALSE;
    }
    return TRUE;
  }  .
```

4 Использование ресурса меню

Окно с рамкой кроме системного меню может содержать и меню, спроектированное и настроенное пользователем. Такое меню является ресурсом окна и сообщения, связан-

ные с выбором пунктов меню, обрабатываются обработчиком окна. Если это главное окно, то они обрабатываются обработчиком главного окна. Само меню может рассматриваться как определенным образом организованная совокупность командных кнопок, которые представляют пункты меню. В основном, это кнопки двух типов - MENUITEM и POPUP. Кнопка типа POPUP определяет пункт главного меню, автоматически вызывающий выпадающее подменю, которое может содержать подпункты типа MENUITEM и POPUP. Кнопка типа MENUITEM определяет конечный пункт меню. При его выборе генерируется сообщение WM_COMMAND, параметр сообщения WPARAM wParam содержит ID выбранного пункта меню. Соответственно для его обработки используются секции в обработчике

```
LRESULT CALLBACK WndProc (HWND hWnd, UINT Message,
                                  WPARAM wParam,  LPARAM lParam)
{
    …….
    switch (Message)
    {
    …….
    case WM_COMMAND:  // сообщение от меню
            switch (wParam)  // параметр сообщения – ID выбранного пункта меню
            {
            case IDM_< КонечныйПунктМеню >:
                    …….
                    break;
    …….
            case IDM_< КонечныйПунктМеню >:
                    …….
                    break;
            …….
                default:
                    return DefWindowProc(hWnd, messg, wParam, lParam);
            }
            break;
            default:
                    return (DefWindowProc(hWnd, messg, wParam, lParam));
    }
    return 0;
}
```

При текстовом описании меню соответственно могут использоваться два типа операторов: MENUITEM и POPUP. Оператор MENUITEM определяет конечный пункт меню, а оператор POPUP — выпадающее меню. Операторы имеют следующий формат:

MENUITEM "*НазваниеПункта*", ID*_Пункта* [, *ОпцииПункта*]
POPUP " *НазваниеПункта* " [, *ОпцииПункта*] .

Параметр *НазваниеПункта* задает название пункта меню, например, File или Help. Параметр ID*_Пункта* - уникальное целое значение, идентифицирующее пункт меню, посылаемое приложению при выборе данного пункта. Обычно ID-значения хранятся в

виде констант в библиотечном файле, который затем включается как в программный файл, так и в rc-файл ресурсов. Опции меню и пунктов меню приведены в таблице 16.

Таблица 16 - Опции меню и пунктов меню

Опция	**Использование**
CHECKED	(опция пункта) для размещения рядом с пунктом меню отметки выбора пункта (для пунктов меню верхнего уровня не используется)
GRAYED	(опция пункта) для пометки пункта меню как не активного (серым, “бледным” цветом). Пункт не может быть выбран пользователем
HELP	(опция пункта) пункт меню может быть связан с командой вызова помощи (применяется только с пунктами типа MENUITEM)
INACTIVE	(опция пункта) пункт меню выводится в списке меню, но не может быть выбран в данных обстоятельствах
MENUBREAK	(опция пункта) то же, что и MENUBARBREAK, но без использования разделительной черты
MENUBARBREAK	(опция пункта) для указания пункта меню, выполняющего роль разделителя других пунктов (в меню верхнего уровня вызывает запись названия нового пункта с новой строки. В выпадающих меню название пункта будет размещено в новом столбце и отделено чертой)
OWNERDRAW	(опция пункта) для указания, что состоянием и изображением пункта меню, включая выделенное, неактивное и отмеченное состояния, отвечает владелец меню
POPUP	(опция пункта) для указания типа пункта меню как POPUP. При выборе этого пункта выводится список пунктов подменю
DISCARDABLE	(опция меню) для указания, что меню может быть удалено из памяти, если больше не используется
FIXED	(опция меню) для указания, что меню постоянно находится в памяти
LOADONCALL	(опция меню) для указания, что меню загружается при обращении

Ниже приведен каркас обработки сообщений меню (обработчик главного окна с пользовательским меню) на примере ТКП типа Hello. В нем поддерживается меню с пунктами File-Exit (выход из приложения) и Help–About (вызов диалогового окна со справочной информацией).

```
LRESULT CALLBACK WndProc( HWND hWnd, UINT message,
                                    WPARAM wParam, LPARAM lParam)
{
    int wmId, wmEvent;
    PAINTSTRUCT ps;
    HDC hdc;
    switch (message)
    {
    case WM_COMMAND:
            wmId    = LOWORD(wParam);
            wmEvent = HIWORD(wParam);
            switch (wmId)
            {
```

```
            case IDM_ABOUT:
                    DialogBox( hInst, ( LPCTSTR) IDD_ABOUTBOX, hWnd,
                                        ( DLGPROC ) About );
                     break;
            case IDM_EXIT:
                    DestroyWindow(hWnd);
                    break;
            default:
                    return DefWindowProc( hWnd, message, wParam, lParam);
            }
            break;
      case WM_PAINT:
            hdc = BeginPaint(hWnd, &ps);
            //  <ФРАГМЕНТ_ПОЛЬЗОВАТЕЛЯ>
            EndPaint(hWnd, &ps);
            break;
      case WM_DESTROY:
            PostQuitMessage(0);
            break;
      default:
            return DefWindowProc(hWnd, message, wParam, lParam);
      }
   return 0;
   } .
```

ПРИМЕР. *Разработать приложение на базе ТКП с окном с рамкой в качестве главного, содержащее простейшее пользовательское меню.*

Интерфейсные формы. Здесь два пункта типа POPUP – ВВОД, ВЫВОД, четыре пункта типа MENUITEM – Строка 1, Строка 2 (подпункты пункта ВВОД), Строка 1, Строка 2 (подпункты пункта ВЫВОД), соответственно с IDM_inString1, IDM_inString2, IDM_outString3, IDM_outString4. Примерный вид интерфейса показан на рисунках 27, 28.

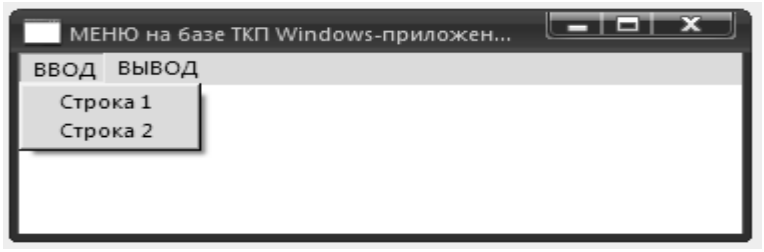


Рисунок 27 - Фрагмент интерфейса 1

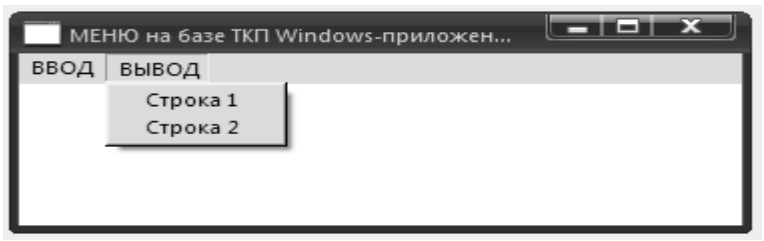


Рисунок 28 - Фрагмент интерфейса 2

Обрабатываемые сообщения. Это сообщения, адресуемые главному окну:

- сообщение WM_PAINT – здесь не используется;
- сообщение WM_DESTROY при закрытии окна. Действие – завершение работы приложения посылкой сообщения на завершение командой PostQuitMessage(0);
- сообщения WM_COMMAND от конечных пунктов меню, в том числе WM_COMMAND:IDM_inString1,WM_COMMAND:IDM_inString2, WM_COMMAND: IDM_outString3, WM_COMMAND: IDM_outString4.

5 Использование стандартных диалоговых окон

Стандартные диалоговые окна позволяют автоматизировать типовые действия пользователя, такие как просмотр стуктуры папок компьютера, поиск, открытие, сохранение файлов и т.д.

Функции, обеспечивающие работу с типовыми диалоговыми окнами, подключаются с помощью заголовочного файла commdlg.h.

Для навигации по структуре папок и выбора нужного файла для открытия используется функция GetOpenFileName, поддерживающая окно типа “Открыть” (рисунок 29), а для сохранения файла используется функция GetSaveFileName, поддерживающая окно типа “Сохранить как”.

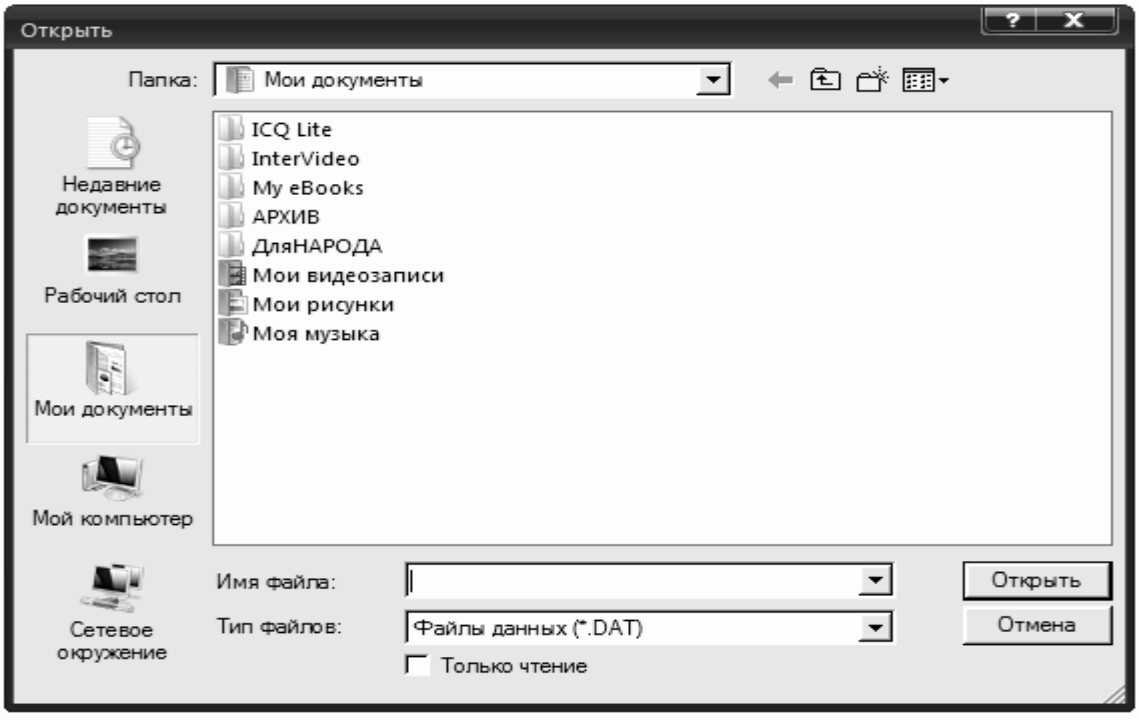


Рисунок 29 - Окно “Открыть”

Прототип функции GetOpenFileName

BOOL GetOpenFileName(LPOPENFILENAME *СсылкаНаСтруктуру*) .

Здесь *СсылкаНаСтруктуру* – указатель типа LPOPENFILENAME на структуру типа OPENFILENAME.

При запуске функции в соответствии с установками структуры типа OPENFILE-NAME инициализируется, визуализируется и активизируется указанное диалоговое окно, которое ждет выбора пользователя. После сделанного выбора функция автоматически завершает свою работу, закрывает окно и заполняет указанную структуру, которая и содержит информацию о пользовательском выборе файла.

Структура типа OPENFILENAMEW, содержащая информацию для инициализации окна

```
typedef struct tagOFN
 {
    DWORD lStructSize;
    HWND hwndOwner;
    HINSTANCE hInstance;
    LPCSTR lpstrFilter;
    LPSTR lpstrFile;
    DWORD nMaxFile;
    LPSTR lpstrFileTitle;
    LPCSTR lpstrTitle;
    DWORD Flags;
    …
}  OPENFILENAME  .
```

Основные поля описаны ниже. Здесь:

- lStructSize - размер структуры;
- hwndOwner - дескриптор родительского окна (NULL);
- lpstrFilter – указатель строки, буфера, содержащего определенным образом форматированные пары строк, задающие фильтры окна;
- lpstrFile – указатель на строку, содержащую имя файла, которое зафиксировано в окошке редактирования диалогового окна (или NULL). После успешного завершения функций GetOpenFileName или GetSaveFileName это указатель буфера, содержащего полное имя выбора пользователя;
- nMaxFile – размер буфера, строки lpstrFile;
- Flags – флаги (по завершении работы идентифицируют пользовательский ввод).

Соответственно при использовании типовых окон (например, типа “Открыть”, “Сохранить как”) необходимо:

1. До начала работы подготовить структуру типа OPENFILENAME. Например,

```
char ИмяФайла [128];
char Фильтр [ ] = "Файлы данных (*.doc) \0 *.dat\0 Все файлы (*.*)\0*.*\0";

OPENFILENAME СтруктураВыбранногоФайла;
memset(&СтруктураВыбранногоФайла, 0, sizeof(OPENFILENAME));
СтруктураВыбранногоФайла.lStructSize = sizeof(OPENFILENAME);
СтруктураВыбранногоФайла.hwndOwner = ДескрипторРодительскогоОкна;
СтруктураВыбранногоФайла.lpstrFilter = Фильтр;
СтруктураВыбранногоФайла.lpstrFile = ИмяФайла;
СтруктураВыбранногоФайла.nMaxFile = sizeof(ИмяФайла);
СтруктураВыбранногоФайла.Flags=  OFN_PATHMUSTEXIST  |  OFN_FILE MUSTEXIST;  .
```

2. Выполнить функцию для запуска окна и получения выбора пользователя. Например,

GetOpenFileName(&*СтруктураВыбранногоФайла*) .

3. Если выбор состоялся, то обработать его, использовав полученное имя файла. Например,

```
if ( GetOpenFileName (&СтруктураВыбранногоФайла ) )
{
        ОБРАБОТАТЬ_ФАЙЛ   ИмяФайла
}  .
```

ПРИМЕР. *Создать приложение с окном с рамкой. По сообщению WM_ LBUTTONDOWN выводить окно типа “Сохранить как” для получения пользовательского указания места в файловой системе компьютера и имени файла. По сообщению WM_RBUTTONDOWN выводить окно типа “Открыть” для выбора открываемого файла.*

Вариант текста приложения приведен ниже

```
#include <commdlg.h>
…
char szFileName[128];
char szFilter[] = "Файлы данных (*.doc) \0*.dat\0Все файлы (*.*)\0*.*\0";
LRESULT CALLBACK   WndProc(HWND, UINT, WPARAM, LPARAM);
…
int  WINAPI WinMain (…) {   … }
LRESULT CALLBACK WndProc (HWND hWnd, UINT Message,
                                        WPARAM wParam, LPARAM lParam)
{
    HDC hdc;
    PAINTSTRUCT ps;
    OPENFILENAME ofn;
    switch (Message)
    {
    case WM_RBUTTONDOWN:
            memset(&ofn, 0, sizeof(OPENFILENAME));
            ofn.lStructSize = sizeof(OPENFILENAME);
            ofn.hwndOwner = hWnd;
            ofn.lpstrFilter = szFilter;
            ofn.lpstrFile = szFileName;
            ofn.nMaxFile = sizeof(szFileName);
            ofn.Flags=OFN_PATHMUSTEXIST|OFN_FILEMUSTEXIST;
            if  (GetOpenFileName(&ofn) )
            {
                    //Команды по обработке файла с именем szFileName
                    break;
            }
            else
                    break;
    case WM_LBUTTONDOWN:
            memset(&ofn, 0, sizeof(OPENFILENAME));
```

```
            ofn.lStructSize = sizeof(OPENFILENAME);
            ofn.hwndOwner=hWnd;
            ofn.lpstrFilter = szFilter;
            ofn.lpstrFile = szFileName;
            ofn.nMaxFile = sizeof(szFileName);
            ofn.Flags=OFN_PATHMUSTEXIST|OFN_FILEMUSTEXIST;
            if (GetSaveFileName(&ofn) )
            {
                  //Команды по обработке файла с именем szFileName
                  break;
            }
            else
                  break;
      case WM_PAINT:
             …
      case WM_DESTROY:
            PostQuitMessage(0);
            break;
      default:
            return (DefWindowProc(hWnd, Message, wParam, lParam));
      }
      return 0;
  } .
```

Лекция № 8 Библиотека MFC

1 Особенности mfc-программирования в ОС Windows

Операционная система Windows базируется на понятии виртуальная машина, что обеспечивает независимость пользователя от специфики аппаратных средств системы и деталей реализации программных компонентов предоставляемого программного обеспечения. Доступ к системным ресурсам осуществляется через системные функции, образующие прикладной программный интерфейс – API (Application Programming Interface). Поддержку аппаратно-независимой графики обеспечивает подмножество функций API – интерфейс графического устройства GDI (Graphics Device Interface). Функции GDI дают программисту средства создания и реализации оконных интерфейсов. В настоящее время преимущественно используется 32-разрядная версия API, называемая Win32 и являющаяся надмножеством версии Win16. Существенный момент - Win32 поддерживает 32-разрядную линейную адресацию памяти взамен 16-разрядной сегментированной модели. Соответственно управление памятью системы производится в защищенном режиме на базе виртуальной – линейной, плоской модели памяти в 4 Гб с использованием механизмов страничного скроллинга. Windows обеспечивает мультипрограммность работы системы, ее многозадачность. Поддерживается два типа многозадачности: многозадачность, основанная на процессах и основанная на потоках. Соответственно под его управлением одновременно может выполняться несколько приложений (процессов), а в рамках приложений могут быть определены параллельно выполняемые и при необходимости синхронизируемые участки (потоки, “направления протекания процесса”). Снижение объемов приложений, их живучесть и мобильность также обеспечиваются использованием библиотек динамической загрузки - DLL (Dynamic Link Libraries, или), ко-

торые загружаются в память только в момент, когда к ним происходит обращение, т.е. при выполнении программы.

Оконные приложения функционируют под управлением операционной системы Windows и взаимодействуют с ресурсами системы, с внешним миром посредством механизма пересылки сообщений через Windows о происходящих событиях. Приложение может реагировать на полученное сообщение – обрабатывать его, выполняя определенные в приложении программистом действия. При этом в промежутке между обработкой сообщений приложение бездействует. Оконные приложения используют привычный пользователю, типизированный Windows-интерфейс на базе системы окон. Используются окна с клиентской областью, диалоговые окна различных типов, включая стандартные. Как только окно активизируется (получает фокус), то именно с ним ассоциируются приходящие сообщения. Соответственно их обработкой занимаются специальные функции-обработчики сообщений, ассоциированные с окном.

При разработке приложений, исполняемых в операционной системе Windows, используют два основных подхода. *Первый подход* - процедурная разработка, базирующаяся на алгоритмической декомпозиции предметной области и принципах структурной разработки программ. В Windows это стиль низкоуровневого программирования с непосредственным использованием функций операционной системы. *Второй подход* – объектно-ориентированная разработка (проектирование, программирование, тестирование), базирующаяся на объектной декомпозиции предметной области и принципах построения объектно-ориентированной модели ПО. В Windows это стиль высокоуровневого программирования с использованием библиотек готовых классов. Например, библиотеки MFC (Microsoft Foundation Class Library), применяемой для создания Windows-приложений с графическим интерфейсом. Результатом применения этого стиля в системе программирования Visual Studio являются оконные MFC-приложения как “статичного” так и “динамического” типов. Интерфейсные функции приложений реализуются программированием, основанным на ресурсах. Типовые интерфейсные ресурсы - растровые изображении, пиктограммы, описания меню, диалоговых окон и т.п. Описания ресурсов в процессе реализации интерфейса приложения редактируются специальными инструментальными средствами – редакторами ресурсов, обеспечивающими режим WYSIWYG. Описания ресурсов хранятся в файлах ресурсов.

Характеристика библиотеки. Чтобы облегчить программирование, особенно выполнение стандартных задач, к которым относится и создание многооконных интерфейсов, системы программирования, компиляторы используют специальные библиотеки. Библиотека MFC ориентирована на существенное упрощение работы с прикладным программным интерфейсом (API) Windows, который включает сотни разрозненных API функций, за счет их структуризации. В MFC функции API объединены (путем инкапсуляции, в виде методов) в логически организованное множество классов, что позволяет использовать при программировании средства более высокого уровня, чем обычные вызовы функций. Допускается и смешанное использование. API функции можно использовать в MFC-приложении, написанном на языке C++. Часть классов MFC (классы общего назначения - файлы, строки, время, исключительные ситуации и т.д.) можно использовать как в DOS- так и в Windows-приложениях.

Таким образом, объектно-ориентиро-ванная библиотека MFC – это базовый набор (библиотека) классов, написанных на языке C++ и предназначенных для упрощения и ускорения процесса программирования для ОС Windows. Библиотека включает многоуровневую иерархию классов (около 200 членов). Они поддерживают высокоуровневый стиль разработки Windows-приложений на базе объектно-ориентированного подхода. Практически классы MFC представляют собой каркас, на основе которого можно писать программы для ОС Windows.

Библиотека MFC реализует принцип многократного использования одного и того же кода, позволяя просто наследовать типовые составляющие приложений (наследовать классы). Интерфейс, обеспечиваемый библиотекой, практически независим от конкретных деталей реализации, поэтому программы, написанные на основе MFC, легко адаптируются к новым версиям Windows. Это существенно упрощает программирование, так можно использовать готовые классы MFC для создания объектов в пользовательских приложениях, можно дорабатывать классы MFC, наследуя их и создавая пользовательские классы и иерархии классов.

Технологически процесс программирования в MFC реализуется как “каркасное” программирование. Библиотека MFC, в том числе, обеспечивает технологическую, программную поддержку каркасов архитектуры приложений типа “ОДИНОЧНЫЙ_ДОКУМЕНТ” с интерфейсом SDI (Single Document Interface) и типа “ДОКУМЕНТ-ВИД” с интерфейсом MDI (Multiple Document Interface), образующих ядро программирования в MFC. Первый каркас использует класс ОКНО_С_РАМКОЙ (CFrameWnd) для организации всех функций по обработке данных (документов), включая их хранение, визуализацию. Последний каркас является базовым для построения интерфейсов многодокументных оконных приложений, а соответствующий каркас и архитектура образуют ядро программирования в MFC. Здесь объект обработки разделяется на класс ВИД (CView) и класс ДОКУМЕНТ (CDocument). Соответственно приложение может состоять из одного или нескольких документов - объектов, классы которых являются производными от класса ДОКУМЕНТ. С каждым из документов может быть связан один или несколько обликов - объектов классов, производных от класса ВИД. Класс ВИД обеспечивает, определяет оконный вид, облик, представление документа, а класс ДОКУМЕНТ позволяет представлять более абстрактные объекты, чем документ, понимаемый как объект обработки текстового процессора.

К числу дополняющих библиотек можно отнести: - библиотеку стандартных шаблонов STL (Standard Template Library); - библиотеку шаблонов ActiveX ATL (Microsoft Active Template Library), предназначенную для создания небольших COM-объектов и элементов управления ActiveX; - объектно-ориентированную библиотеку WFC (Windows Foundation Classes) для Windows-приложений на языке Java и др. Библиотека MFC реализована как в виде обычной статично подключаемой библиотеки (Static Library) так и в виде динамически подключаемых библиотек DLL (Shared Dynamic Link Libraries). Считается, что компилятор в оптимизирующем режиме создает MFC-приложения, несущественно отличающиеся по своим характеристикам от аналогичных приложений на языке C++, созданных с непосредственным использованием API функций. Но при существенно меньшем размере кода (в три и более раза). При этом, что немаловажно, большинство частных деталей скрыто от программиста и не требует знания.

Библиотека MFC содержит типовые средства, необходимые для функционирования Windows-приложений. Это, в частности, средства управления сообщениями, окнами, диалоговыми окнами, ресурсами типа меню, кисти, перья, растровые изображения и т.п., средства работы с документами и многодокументным интерфейсом (MDI). Библиотека MFC поддерживает механизм обработки исключительных ситуаций, обеспечивая восстанавливаемость приложений после появления ошибок. MFC также обеспечивает механизм динамического определения типов объектов, что позволяет производить проверку и преобразование типов динамически в процессе выполнения программы.

Характеристика классов библиотеки. Большинство классов библиотеки MFC являются производными от базового класса CObject ("объект") и предназначено для построения компонентов приложения. Подробную информацию о классах, а также полный вариант диаграммы классов MFC можно найти в справочной информации Visual Studio – MSDN. Названия всех классов и шаблонов классов библиотеки MFC начинаются с за-

главной буквы С. Имена переменных, входящих в класс MFC начинаются с префикса m_. Названия методов классов, как правило, начинаются с заглавной буквы. Названия служебных (глобальных) функций библиотеки MFC начинаются с префикса Afx, например, AfxMessageBox. Фрагмент соответствующей диаграммы классов приведен ниже (рисунок 30). Здесь CCmdTarget является основой для формирования архитектуры приложения (двух ее компонентов – интерфейсного класса и класса, управляющего движением и обработкой сообщений). Соответственно у CCmdTarget есть производные классы: - CWinThread позволяет создавать поток приложения, а его производный класс CWinApp - основа приложения; - класс CWnd является базовым для создания разнообразных окон, в том числе, окон с рамкой класса CFrameWnd, диалоговых окон класса CDialog, специализированных окон ЭУ, обликов класса CView; - класс CDocument служит для организации работы с документами.

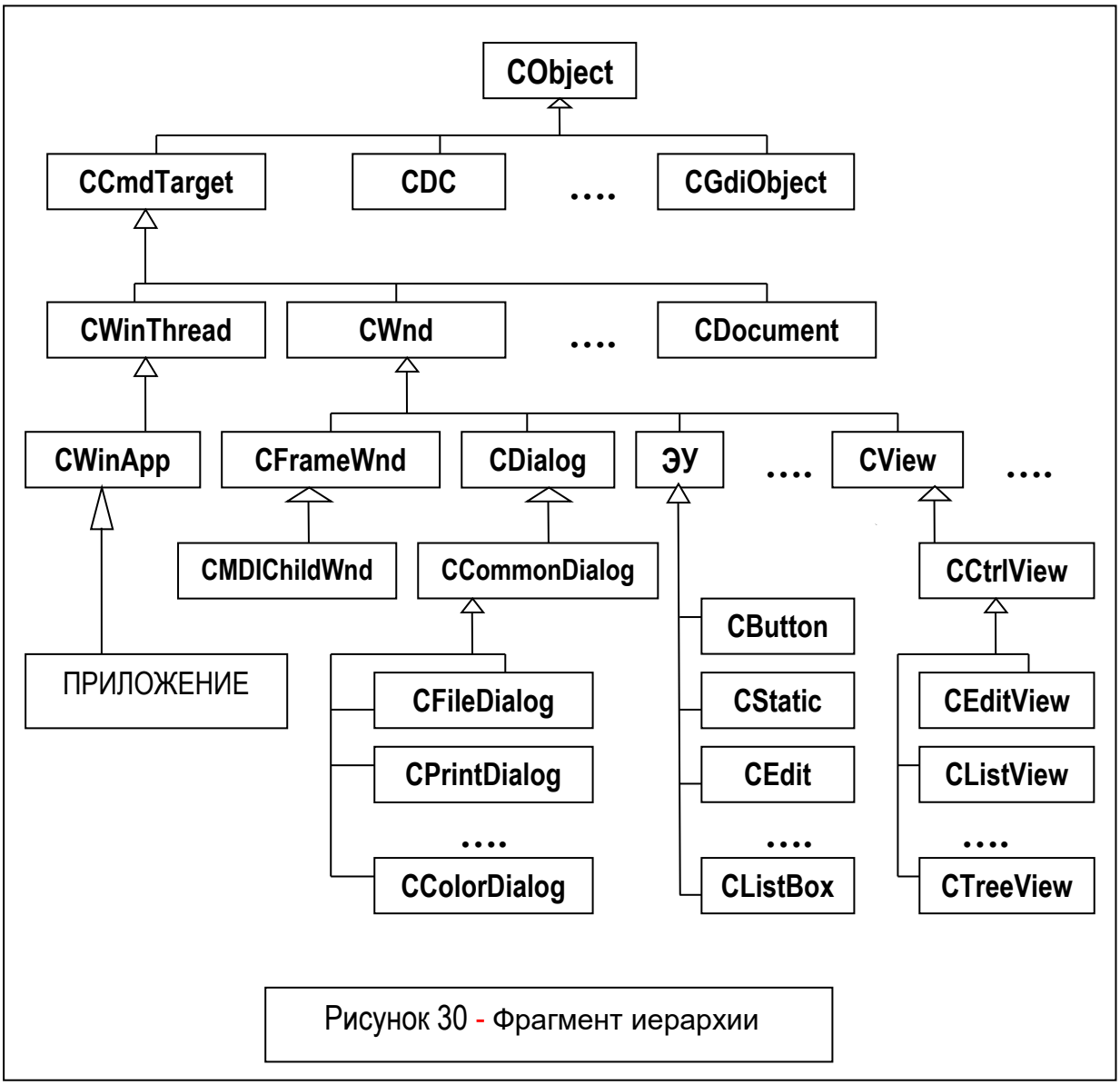


Рисунок 30 - Фрагмент иерархии

С типовым MFC-приложением связывается определяющий его на верхнем уровне объект, принадлежащий классу, производному от класса CWinApp. Оконный интерфейс приложения строится как система взаимодействующих окон различных стилей и типов, производных от класса CWnd. Так для приложений с однодокументной архитектурой интерфейс строится на базе классов CFrameWnd, CDialog, семейства классов элементов

управления (ЭУ) и др. Рамочные окна типа главного окна, масштабируемые и перекрываемые, с меню и без, с клиентской областью, предназначенной для вывода текстовой и графической информации на экран, производятся от класса CFrameWnd. Диалоговые окна интерфейса, являющиеся подложкой для размещения и компоновки различных ЭУ, производятся от класса CDialog. Каждый ЭУ, представляющий собой специфическое окно, производится от соответствующего класса семейства классов элементов управления. Для приложений с многодокументной архитектурой документы представляются как объекты, производные от класса CDocument. С каждым из документов связаны объекты, производные от класса CView (класс "облик ") и определяющие облик документа.

Класс CWnd. Это базовый для создания окон разных типов в MFC-приложениях. Окна MFC-приложений (CWnd-объекты) отличаются организацией от классических Windows-окон, хотя и тесно связаны с ними функционально. Класс CWnd, механизм обработки событий (карта событий) позволили скрыть функцию, обработчик главного окна. Сообщения, автоматически маршрутизируясь на основе карты сообщений, направляются на обработку в соответствующие функции-обработчики класса CWnd. Класс CWnd позволяет создавать производные пользовательские классы и на их основе объекты – пользовательские окна. При этом программист в своих производных классах переописывает (override) обработчики, определенные в классе CWnd, чтобы обрабатывать свои сообщения нужным образом. Такие окна создаются: 1) вызовом конструктора CWnd() для создания CWnd-объекта; 2) вызовом метода Create для создания пользовательского окна, связанного с этим CWnd-объектом. Дескриптор созданного окна хранится в атрибуте m_hWnd (HWND CWnd::m_hWnd).

Класс CWnd содержит методы получения контекста BeginPaint, EndPaint, перерисовки клиентской области Invalidate (перерисовка всей клиентской области), InvalidateRgn, InvalidateRect (перерисовка части клиентской области в пределах заданной прямоугольной области путем ее добавления к текущей области обновления), методы визуализации и обновления окна ShowWindow, UpdateWindow, методы управления таймером SetTimer, KillTimer. Отдельную группу составляют обработчики событий. В том числе, обработчики сообщений инициализации (например, OnInitMenu вызываемый при активизации меню), обработчики системных событий (например, OnSysCommand, вызываемый, когда пользователь выбирает команды системного меню, использует кнопки минимизации-максимизации размеров окна), обработчики общих событий и т.д.

Для создания дочернего (child) окна и связывания его (прикрепления) с CWnd-объектом применяется метод Create, который возвращает ненулевое значение при успешном завершении, и имеет прототип

```
virtual BOOL CWnd::Create
                        (LPCTSTR lpszClassName,
                         LPCTSTR lpszWindowName,
                         DWORD dwStyle,
                         const RECT &rect,
                         CWnd  *pParentWnd,
                         UINT nID,
                         CCreateContext  * Context = NULL)  .
```

Параметры метода:

- lpszClassName – указатель на строку С (с нуль-символом), которая содержит имя класса окна. Этим именем может быть любое имя, зарегистрированное с помощью глобальной функции AfxRegisterWndClass или API функцией RegisterClass. Значение NULL позволяет использовать типовое MFC-окно;

- lpszWindowName - указатель на строку С (с нуль-символом), которая содержит заголовок окна;
- dwStyle – специфицирует атрибуты стиля окна;
- rect - специфицирует размер и положение окна;
- pParentWnd – указатель на родительское окно;
- nID – ID дочернего (child) окна;
- pContext – контекст окна.

ПРИМЕР. Динамическое создание ЭУ типа static в окне класса CMyDlg, путем запуска обработчика окна с именем CMyDlg::OnCreateStatic(). В обработчике использован метод CWnd::Create (вместо метода CStatic::Create)

```
void CMyDlg::OnCreateStatic()
{
    CWnd   *pWnd = new CWnd;
    pWnd -> Create(_T("STATIC"),"Hi",WS_CHILD | WS_VISIBLE,CRect(0,0,20, 20),
                   this, 1234);
}  .
```

Класс CFrameWnd. Класс предоставляет возможности создания объектов-окон с рамками типа overlapped или pop-up, обеспечивающими одно документный интерфейс (SDI). Чтобы создать такое пользовательское окно надо: - создать производный класс; - добавить члены-данные для хранения специфических данных; - настроить карту сообщений окна; - описать обработчики сообщений.

Непосредственное создание, инициализация окон производится методом CFrameWnd::Create, который возвращает ненулевое значение при успешном завершении, и имеет прототип (похожий на прототип метода CWnd::Create)

```
BOOL Create (LPCTSTR        lpszClassName,
             LPCTSTR        lpszWindowName,
             DWORD          dwStyle = WS_OVERLAPPEDWINDOW,
             const RECT&    rect = rectDefault,
             CWnd*          pParentWnd = NULL,
             LPCTSTR        lpszMenuName = NULL,
             DWORD          dwExStyle = 0,
            CCreateContext* pContext = NULL )  .
```

Здесь параметры метода:

- dwStyle – специфицирует атрибуты стиля окна и по умолчанию (значение WS_OVERLAPPEDWINDOW) задает создание перекрывающегося масштабируемого окна с системным меню и системными кнопками;
- rect - специфицирует размер и положение окна, которые по умолчанию (значение rectDefault - свойство класса CFrameWnd::rectDefault типа CRect) определяются ОС Windows автоматически;
- lpszMenuName – определяет имя ресурса-меню, используемого окном (по умолчанию NULL). Если ресурс-меню идентифицируется числовым значением ID, то можно использовать функцию MAKEINTRESOURCE для получения строкового эквивалента;

2 Использование Visual Studio

Интегрированная среда. Средства разработки фирмы Microsoft (Microsoft Developer Studio) - универсальная среда, поддерживающая различные языковые средства разработки, например, Visual C++, Visual J++ и т.д. (здесь рассматривается Visual C++).

Назначение среды – поддержка процесса создания Windows-приложений с использованием библиотеки базовых классов MFC и других библиотек (как статических так и динамических) как “вручную” так и с использованием средств автоматизации. К числу средств автоматизации можно отнести мастера приложений (генераторы каркасов приложений нужной архитектуры для последующей доработки), мастер классов ClassWizard (для создания шаблонов новых производных классов и их методов), редакторы ресурсов (для создания меню, диалоговых окон, элементов управления и т.п.).

При работе с MFC-приложениями можно использовать мастера приложений MFC AppWizard (exe), MFC AppWizard (dll), позволяющие создавать каркасы разных типов статических и динамических (dll) проектов. С их помощью можно создать проект Windows-приложения с однодокументным, многодокументным или диалоговым интерфейсом. Однодокументное приложение предоставляет пользователю возможность работы в любой момент времени только с одним файлом. Многодокументное приложение предоставляет пользователю возможность одновременной загрузки и обработки нескольких документов (каждый в собственном окне). Пользовательский интерфейс диалогового приложения базируется на диалоговом окне, используемом в роли главного окна приложения. Мастер ClassWizard позволяет добавлять в приложение новые классы, созданные на основе базовых классов (как правило, классов, унаследованных от класса CCmdTarget или класса CRecordset), автоматизировать добавление и описание новых методов (функций) классов, методов, обрабатывающих сообщения ресурсов, меню, ЭУ и т.д. Средства AppWizard и ClassWizard для указания принадлежности того или иного объекта (функции, переменной, ключевого слова и т.д.) к библиотеке MFC используют сокращение AFX (от Application FrameworkX - основа приложения, его внутреннее устройство). Например, в процессе генерации они размещают в исходном тексте приложения комментарии следующего вида: //{{AFX_*ИдентификаторСекции* ... //}}AFX_*ИдентификаторСекции*

Эти комментарии образуют секции (блоки) кода программы, которые управляются только средствами MFC (AppWizard, ClassWizard) и не должны меняться пользователем вручную. Идентификаторы некоторых секций приведены в таблице ниже.

Таблица 17 - AFX-Секции

Секция	Назначение
//{{AFX_DATA //}}AFX_DATA	описание (объявление) элементов данных, используемых в классах - диалоговые окна
//{{AFX_DATA_INIT //}}AFX_DATA_INIT	описание инициализации элементов данных (в файле реализации классов - диалоговые окна)
//{{AFX_DATA_MAP //}}AFX_DATA_MAP	описание макрокоманд DDX, предназначенных для связывания элементов данных класса и органов управления диалоговых окон (в файле реализации классов - диалоговые окна)
//{{AFX_MSG //}}AFX_MSG	описание методов классов, которые предназначены для обработки сообщений
//{{AFX_MSG_MAP //}}AFX_MSG_MAP	описание макрокоманд таблицы сообщений

//{{AFX_VIRTUAL //}}AFX_VIRTUAL	описание переопределенных виртуальных методов класса

Настройка Visual Studio. В пункте главного меню ГМ-Project, предназначенном для управления открытыми проектами, можно задать конкретные настройки проекта (языковые настройки, режимы транслирования, оптимизации, отладки, компоновки, использования библиотек) командой ГМ-Project-Settings…. При этом в появившемся окне Project Settings на вкладке General надо подключить библиотеку MFC с указанием технологии подключения. В пункте главного меню ГМ-Tools командой Options необходимо вызвать окно Options и на вкладке Directories (Пути) при необходимости уточнить местоположение библиотек, папок Visual Studio, файлов пользователя и порядок их просмотра при работе системы. Соответствующие окна представлены ниже (рисунки 31-33).

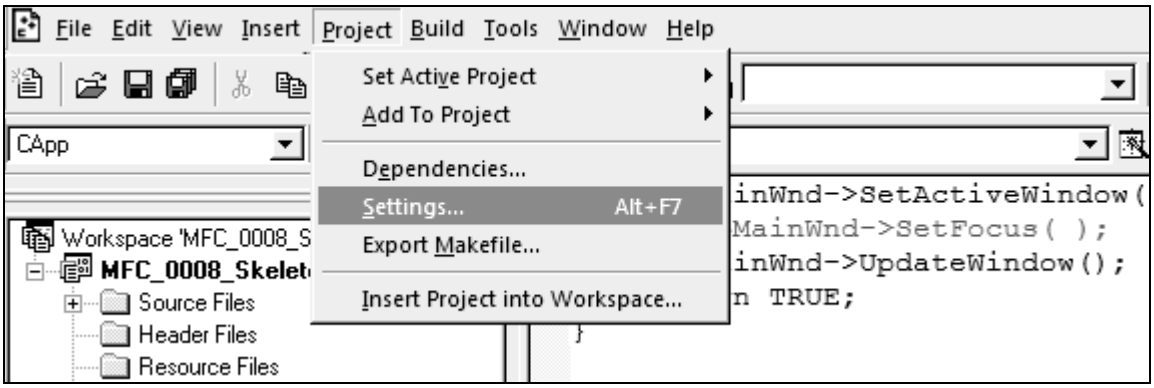


Рисунок 31 - Окно Project

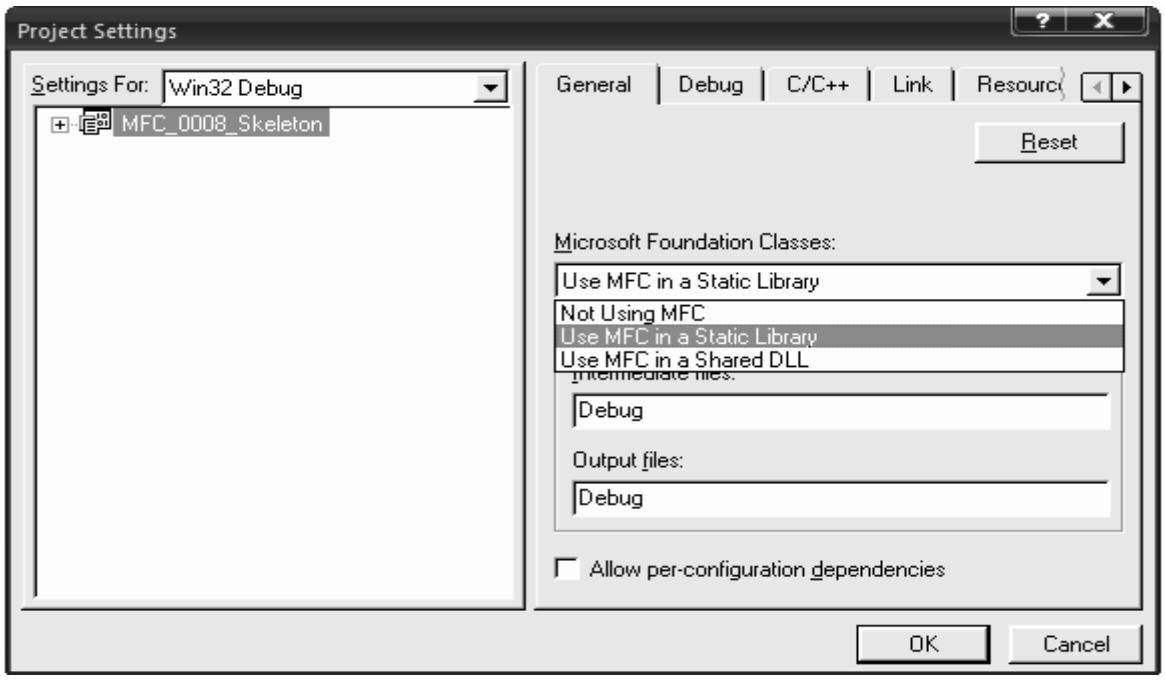


Рисунок 32 - Окно Project Settings

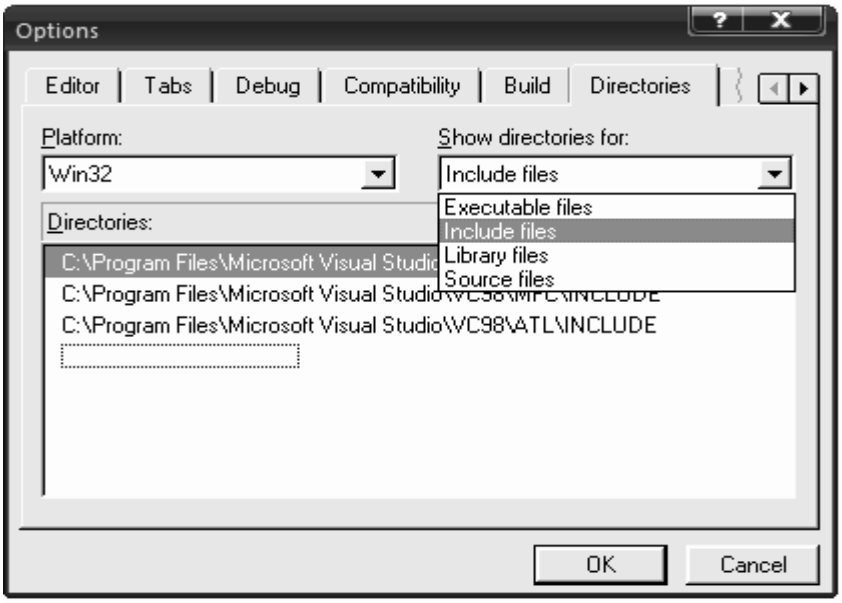


Рисунок 33 - Окно Options

Базовые классы mfc-приложения. У всех Windows-приложений фиксированная структура, определяемая функцией WinMain. Структура приложения, построенного из объектов классов библиотеки MFC, является еще более определенной. В таком приложении используются как минимум объекты двух типов классов, служащих как для организации оконного, графического интерфейса и обработки сообщений, так и для реализации функций самого приложения по запуску, завершению, восприятию и диспетчированию движением сообщений. Так однодокументное MFC-приложение, как правило, строится на базе двух классов библиотеки MFC.

Это класс CFrameWnd (класс ОКНО_С_РАМКОЙ), который служит базовым для создания интерфейса приложения (например окна приложения – класс моеОКНО). Окно с рамкой (типа главного окна) - это окно, внутри которого как бы размещены все остальные окна приложения.

Второй класс CWinApp (класс ПРИЛОЖЕНИЕ_WINDOWS) служит базовым для создания самого приложения (например, класса моеПРИЛОЖЕНИЕ). Он является производным в том числе и от класса CCmdTarget и наследует от него таблицу (карту) сообщений (Message Map) - набор макрокоманд, позволяющий сопоставить сообщения Windows и команды метода класса.

Тогда скелет приложения выглядит как

```
Class  моеОКНО: class CMainWin  { …};
class  моеПРИЛОЖЕНИЕ: public CWinApp  { …};
```

Объект класса CWinApp (соответственно и объект производного от него класса, например, моеПРИЛОЖЕНИЕ) является сердцевиной приложения, отвечает за создание всех остальных объектов и обработку очереди сообщений. Он является глобальным и создается еще до начала работы функции WinMain. Работа функции WinMain заключается в последовательном вызове двух методов объекта класса CWinApp: InitInstance и Run. В терминах сообщений можно сказать, что WinMain посылает объекту сообщение InitInstance, которое приводит в действие метод InitInstance. Получив сообщение InitInstance, объект (моеПРИЛОЖЕНИЕ) создает внутренние объекты приложения. Процесс создания выглядит как последовательное порождение одних объектов другими.

Набор объектов, порождаемых в начале этой цепочки, определен структурой MFC. Затем сообщение Run - приводит в действие метод Run. В результате объект (моеПРИЛОЖЕНИЕ) начинает циклически выбирать сообщения из очереди и инициировать обработку сообщений обработчиками приложения.

Объект класса CFrameWnd (соответственно и объект производного от него класса, например, моеОКНО) имеет графический образ на экране, с которым может взаимодействовать пользователь. Это интерфейсный объект, связанный с Windows-окном. У него есть главная рамка и с ним могут быть связаны объекты документ и облик. При этом документ содержит данные приложения, облик организует представление этих данных на экране, а окно главной рамки содержит все остальные окна приложения. Когда приходит сообщение (пользователь, например, выбирает команду меню главного окна и возникает командное сообщение), то объектом класса CWinApp (моеПРИЛОЖЕНИЕ) они отправляются сначала объекту главная рамка, а затем объектам документ и облик, информируя их о событии, о пришедшей от пользователя команде.

Создание стиля окна. Обработка сообщений. Как правило, все происходящие события и соответствующие сообщения адресуются активному окну приложения, точнее специальным функциям, обработчикам (программам обратного вызова), выполняющим их обработку. Сообщения несут в своих атрибутах всю информацию о происшедшем событии, необходимую для организации их обработки. Стандартные имена сообщений (см. файл windows.h) формируются по правилам <ИдентификаторСообщения> ::= WM_<НазваниеСообщения>.

В MFC есть набор предопределенных обработчиков, которые являются членами класса CWnd и могут быть переопределены в приложении, где пользователь и задает алгоритм обработки. Имена обработчиков формируются по правилам

<ИмяОбработчика> ::= On<НазваниеСообщения*> ,

где НазваниеСообщения* конструируется как составное имя, образованное без символов подчеркивания.

Прототип функции-обработчика

afx_msg void <ИмяОбработчика> ([СписокПараметров]).

Сам обработчик имеет вид

```
afx_msg void  моеОКНО:: <ИмяОбработчика> ([СписокПараметров ])
{
    CPaintDC dc(this); или CClientDC dc(this);
    <ФрагментПользователя>
};  .
```

Для указания сообщений, которые должно обрабатывать приложение, используется макрокоманда включения. Это реализуется описанием (макрокоманда DECLARE_MESSAGE_MAP) очереди сообщений (карты подключения) окна вида

```
BEGIN_MESSAGE_MAP(моеОКНО, CFrameWnd)
    Список макрокоманд включения
END_MESSAGE_MAP()   .
```

Макрокоманда включения формируется по правилам

ON_< ИдентификаторСообщения >([СписокПараметров]).

Например, для сообщения LBUTTONDOWN получим ИдентификаторСообщения WM_LBUTTONDOWN, ИмяОбработчика OnLButtonDown, макрокоманда включения ON_WM_LBUTTONDOWN(), прототип функции-обработчика

afx_msg void OnLButtonDown(UINT flags, CPoint loc).

Однако для сообщения COMMAND ИдентификаторСообщения WM_COMMAND, ИмяОбработчика OnCommand, макрокоманда включения ON_COMMAND (*ИмяРесурса*, *ИмяОбработчика*) !

Таким образом, для того, чтобы конкретное окно (например, описанное в классе моеОКНО) реагировало на сообщения, необходимо для каждого из сообщений:

- описать в классе окна соответствующий обработчик;
- включить чувствительность окна к этому типу сообщений.

Примерная структура описаний приведена ниже

```
class моеОКНО : public CFrameWnd
{
  public:
       моеОКНО();
       afx_msg void <ИмяОбработчика> ([СписокПараметров ]);
        ….
        afx_msg void <ИмяОбработчика> ([СписокПараметров ]);
        DECLARE_MESSAGE_MAP()
};

моеОКНО :: моеОКНО()
{
     Create(NULL,”НазваниеОкна”);
};

afx_msg void моеОКНО ::<ИмяОбработчика> ([СписокПараметров ])
{
     CPaintDC dc(this); или CClientDC dc(this);
     <ФрагментПользователя>
};

…..

BEGIN_MESSAGE_MAP(моеОКНО,CFrameWnd)
     ON_< ИдентификаторСообщения >([СписокПараметров ])
      ……
END_MESSAGE_MAP()   .
```

Например, для обработки сообщений WM_PAINT, WM_LBUTTONDOWN

```
class CMainWin : public CFrameWnd
{
public:
     CMainWin();
     afx_msg void OnPaint();
     afx_msg void OnLButtonDown(UINT flags, CPoint loc);
     DECLARE_MESSAGE_MAP()
};

CMainWin::CMainWin()
```

```
{
    Create(NULL, "Пример");
}

BEGIN_MESSAGE_MAP(CMainWin, CFrameWnd)
    ON_WM_PAINT()
    ON_WM_LBUTTONDOWN()
END_MESSAGE_MAP()

afx_msg void CMainWin::OnPaint()
{
     CPaintDC   TheDC(this);
     ……
};

afx_msg void CMainWin::OnLButtonDown(UINT flags, CPoint loc)
{
          ……
};   .
```

Создание класса приложения. Для создания пользовательского класса приложения (например, класса моеПРИЛОЖЕНИЕ) в качестве базового используется класс MFC – приложение CWinApp. В создаваемом классе необходимо описать метод InitInstance (virtual BOOL CWinApp:: InitInstance( )), выполняющий роль конструктора и отвечающий за инициализацию приложения. В нем используется свойство класса CWinThread - m_pMainWnd (CWnd * CWinThread:: m_pMainWnd) для указания на CWnd-объект (окно), т.е. на интерфейсный объект пользовательского класса, принадлежащий приложению.

В методе InitInstance вначале динамическим образом конструируется интерфейсный объект пользовательского класса (например, моеОКНО). Его адрес сохраняется в переменной m_pMainWnd. Затем объект (окно) выводится методом CWnd::ShowWindow(m_nCmdShow) и обновляется методом CWnd::UpdateWindow ( ). Примерный текст описания класса приведен ниже

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

BOOL CApp :: InitInstance ( )
{
    m_pMainWnd = new моеОКНО;
    m_pMainWnd -> ShowWindow ( m_nCmdShow );
    m_pMainWnd -> UpdateWindow ( );
    return TRUE;
}   .
```

3 Структура типового MFC-приложения

Общая структура типового MFC-приложения, реализованного в виде проекта с одним модулем на базе двух пользовательских классов, производных от CFrameWnd (например, моеОКНО) и CWinApp (например, моеПРИЛОЖЕНИЕ), представлена ниже. Подключение заголовка <iostream> делает видимым описание стандартного пространства имен std и позволяет использовать команду using namespace std.

```
#include  <afxwin.h>
#include  <iostream>
[ < Команды_препроцессора > ]
using namespace std;
[ < Прототипы_прикладных_функций > ]
[ < Описания_глобальных_объектов > ]
< Описание_класса_ГЛАВНОЕ_ОКНО_ПРИЛОЖЕНИЯ_ПОЛЬЗОВАТЕЛЯ >
< Описание_класса_ПРИЛОЖЕНИЕ_ПОЛЬЗОВАТЕЛЯ >
< Создание_экземпляра_класса_ПРИЛОЖЕНИЕ_ПОЛЬЗОВАТЕЛЯ >  .
```

Более детально структура типового MFC-приложения описана ниже

```
#include  <afxwin.h>
#include  <iostream>
using namespace std;
//=== Описания глобальных объектов ============================
//=== Прототипы_прикладных_функций ============================
//=== Описание класса ГЛАВНОЕ_ОКНО_ПРИЛОЖЕНИЯ_ПОЛЬЗОВАТЕЛЯ
class моеОКНО: public CFrameWnd
{
 public:
     CMainWin ( );
     afx_msg void On<ИмяСообщения>( < ПараметрыСообщения > );
    ……….
     afx_msg void On<ИмяСообщения>( < ПараметрыСообщения > );
     afx_msg void OnPaint ( );
     DECLARE_MESSAGE_MAP( )
};

BEGIN_MESSAGE_MAP(CMainWin,CFrameWnd)
     ON_WM_<ИмяСообщения> ( < ПараметрыСообщения > )
      ……..
     ON_WM_<ИмяСообщения> ( < ПараметрыСообщения > )
     ON_WM_PAINT( )
END_MESSAGE_MAP( )

моеОКНО:: моеОКНО ( )
{
     Create (NULL, " КАРКАС MFC-ПРИЛОЖЕНИЯ ");
};
```

```
afx_msg void On<ИмяСообщения>( < ПараметрыСообщения > )
{
      CClientDC  dc (this);
      < ФрагментПользователя >
};
       ……

afx_msg void CMainWin :: OnPaint ( )
{
      CPaintDC  dc (this);
      < ФрагментПользователя >
};

//=== Описание_класса_ПРИЛОЖЕНИЕ_ПОЛЬЗОВАТЕЛЯ ========
class моеПРИЛОЖЕНИЕ: public CWinApp
{
 public:
     BOOL InitInstance ( );
};

BOOL CApp :: InitInstance ( )
{
      m_pMainWnd = new моеОКНО;
      m_pMainWnd -> ShowWindow ( m_nCmdShow );
      m_pMainWnd -> UpdateWindow ( );
      return TRUE;
};   .
```

Структура и код mfc-приложения. Большинство однотипных MFC-приложений имеют одинаковую базовую структуру, называемую скелетом, каркасом приложения. Соответственно в Visual Studio существуют мастера приложений, которые автоматически генерируют тексты каркасов приложений заданных типов. Простейший, типовой каркас MFC-приложения (ТКП) функционально совпадает с типовым каркасом Windows-приложением и выводит на экран окно с рамкой, клиентской областью, содержащее системное меню, системные кнопки (закрытия, минимизации-максимизации окна). Окно масштабируемое, перекрывающееся (с автоматической посылкой сообщений на перерисовку при искажении), с возможностью перемещения (при этом оно автоматически перерисовывается) и изменения размеров. Без пользовательской реакции на сообщения (события).

Базируется на двух пользовательских классах производных от CFrameWnd (например, пользовательский класс моеОКНО) и CWinApp (например, пользовательский класс моеПРИЛОЖЕНИЕ). Для получения контекста устройства – создания соответствующего объекта используется класс CDC и его производные классы. Фрагмент иерархии классов MFC, используемых при создании каркасов приложений, приведен ниже (рисунок 34).

Отношения классов каркаса MFC-приложения приведены ниже (рисунок 35). Указанные классы связаны отношением типа “агрегация”, т.к. класс моеПРИЛОЖЕНИЕ включает объект класса моеОКНО (точнее указатель m_pMainWnd на объект класса моеОКНО). Кратность отношения 1:1. Кроме этого, в методе InitInstance класс моеПРИЛОЖЕНИЕ использует методы класса моеОКНО для его запуска, визуализации,

обновления. Поэтому класс моеПРИЛОЖЕНИЕ и использует класс моеОКНО (существует отношение типа “зависимость”).

Подобные каркасы можно получить автоматически, используя мастер MFC AppWizard(exe) в таких режимах как Single document, Multiple document, Dialog based и т.п. Однако такие автоматические каркасы включают большое количество дополнительных ресурсов, компонентов, средств.

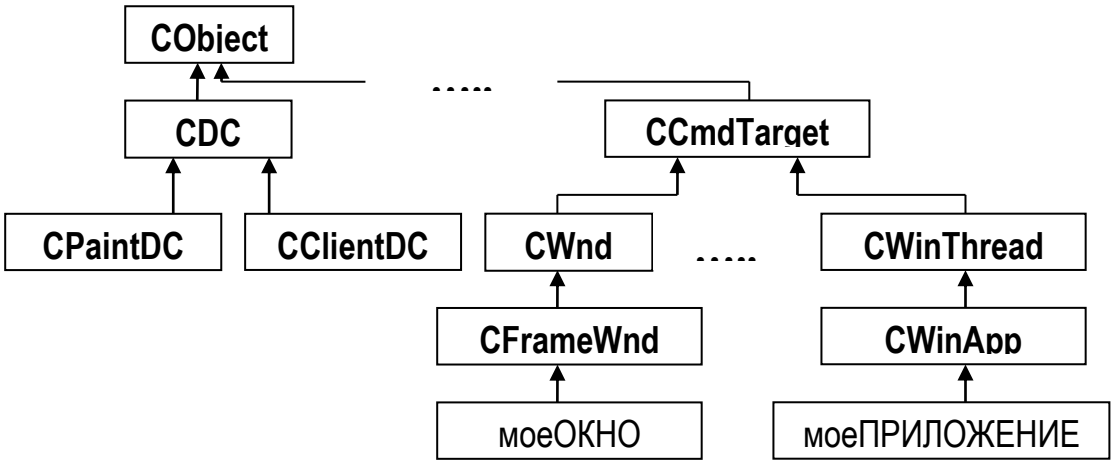


Рисунок 34 - Фрагмент MFC для каркаса

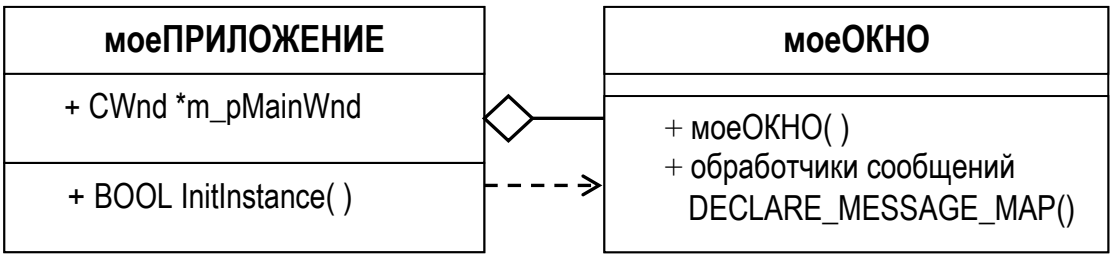


Рисунок 35 - Классы каркаса

Ниже рассматривается и комментируется примерный текст ТКП.

```
#include    <afxwin.h>
//#include  <afxext.h>
//#include  <afxdisp.h>
//#include  <afxdtctl.h>
//#ifndef   _AFX_NO_AFXCMN_SUPPORT
//#include  <afxcmn.h>
//#endif   // _AFX_NO_AFXCMN_SUPPORT
```

```
#include  <iostream>
using namespace std;

    // ===== класс главного ОКНА приложения ==================
    class WINDOW : public CFrameWnd
    {
    public:
        // Конструктор окна
          WINDOW ();
        // Прототип обработчика сообщения WM_LBUTTONDOWN
        // afx_msg void  OnLButtonDown (UINT flags, CPoint loc);
        // Макрос объявления очереди сообщений окна
          DECLARE_MESSAGE_MAP()
    };

    // ===== конструктор окна - Создание окна
    WINDOW:: WINDOW ()
    {
        Create ( NULL,   // имя стиля окна
                   "Главное окно MFC-приложения",  // заголовок окна
                  // DWORD Style = WS_OVERLAPPEDWINDOW,  // стиль окна
                  // const RECT &XYsize =  rectDefault,  // положение и размеры окна
                  // CWnd *Parent = NULL,  // окно-предок
                  // LPCSTR MenuName = NULL,  // имя главного меню
                  // DWORD ExStyle = 0,  // спецификатор расширенного стиля
                  // CCreateContext  * Context  = NULL);  // дополнительная информация
    }

    // ===== метод Обработчик сообщения
    //          нажата левая клавиша “мыши”  WM_LBUTTONDOWN
    // afx_msg void WINDOW::OnLButtonDown (UINT flags, CPoint loc)
    //  { < ФРАГМЕНТ_ПОЛЬЗОВАТЕЛЯ > };

    // ===== очередь сообщений главного окна
    BEGIN_MESSAGE_MAP(WINDOW, CFrameWnd)
          // Макрокоманды включения обрабатываемых сообщений
          // ON_WM_LBUTTONDOWN()
          //  …….
    END_MESSAGE_MAP()

    // ===== класс ПРИЛОЖЕНИЕ ==================
    class APPLICATION : public CWinApp
    {
    public:
       // Конструктор приложения
```

```
    BOOL InitInstance();
};

// ===== конструктор приложения - Инициализации приложения
BOOL APPLICATION::InitInstance ()
{
    // Создание объекта класса WINDOW
      m_pMainWnd = new  WINDOW;
    // Визуализация окна в заданном при запуске приложения стиле m_nCmdShow
      m_pMainWnd -> ShowWindow ( m_nCmdShow );
    // Посылка сообщения о необходимости обновить содержимое окна
      m_pMainWnd -> UpdateWindow ();
      return TRUE;
}

// ===== создание и запуск экземпляра приложения ====================
APPLICATION    TheApplication; .
```

Здесь заголовочный файл <afxwin.h> используется для подключения стандартных компонентов MFC; <afxext.h> используется для подключения расширений MFC; <afxdisp.h> - для подключения классов автоматизации MFC; <afxdtctl.h> - для подключения MFC-поддержки общих элементов управления Internet Explorer 4; <afxcmn.h> - для подключения MFC-поддержки общих элементов управления Windows.

В качестве имени стиля окна здесь указан NULL, что соответствует стандартному стилю MFC-окна с клиентской областью, с рамкой. По умолчанию параметр стиля окна (таблица ЧЧЧ) задается как WS_OVERLAPPEDWINDOW, что означает создание перекрывающегося окна на базе стилей WS_OVERLAPPED, WS_THICKFRAME и WS_SYSMENU, WS_CAPTION, WS_MINIMIZEBOX, WS_MAXIMIZEBOX, т.е. масштабируемого окна с системными меню и кнопками. По умолчанию положение и размеры окна задаются как rectDefault, т.е. определяется автоматически системой Windows.

Построение каркаса mfc-приложения в Visual Studio. Используя мастер каркасов MFC AppWizard(exe), можно получить достаточно сложные каркасы (заготовки) для создания приложений различных типов. Например, можно создать каркасы типа Single, Multiple, Dialog Based documents. Но в целом все они повторяют структуру типового каркаса MFC-приложения.

Для создания MFC-приложения на базе ТКП необходимо: 1) создать ТКП; 2) добавить в ТКП все необходимые компоненты, исходя из назначения приложения.

Для создания ТКП надо: 1) создать “пустой” каркас мастером Win32 Application – выполнить команду File-New, в окне New на вкладке Projects выбрать мастер Win32 Application, задать режим “An empty project”; 2) создать “пустой” файл *.cpp – выполнить команду File-New, в окне New на вкладке Files выбрать тип файла C++ Source File, задать его (пользовательское) имя; 3) открыть указанный файл и скопировать в него текст ТКП; 4) включить в главном меню в пункте Project-Settings, на вкладке General режим – “use MFC in Static library”.

Таблица 18 - Значения параметра dwStyle функции Create

Стиль	Описание
WS_BORDER	создание окна с рамкой
WS_CAPTION	добавление к окну с рамкой заголовка
WS_CHILD	создание дочернего окна. Не используется со стилем WS_POPUP
WS_CLIPCHILDREN	при создании родительского окна запрещает его рисование в области, занятой любым дочерним окном
WS_CLIPSIBLINGS	(только со стилем WS_CHILD) не использовать при получении данным окном сообщения о перерисовке области, занятые другими дочерними окнами. Иначе разрешается рисовать в клиентской области другого дочернего окна
WS_DISABLED	создание первоначально неактивного окна
WS_DLGFRAME	создание окна с двойной рамкой без заголовка
WS_GROUP	(только в диалоговых окнах) стиль указывается для первого элемента управления в группе элементов, пользователь перемещается между ними с помощью клавиш-стрелок
WS_HSCROLL	создание окна с горизонтальной полосой прокрутки
WS_MAXIMIZE WS_MINIMIZE	создание окна максимального (минимального) размера
WS_MAXIMIZEBOX WS_MINIMIZEBOX	создание окна с кнопкой максимизации (минимизации)
WS_OVERLAPPED	создание перекрывающегося окна
WS_OVERLAPPED WINDOW	создание перекрывающегося окна на базе стилей WS_OVERLAPPED, WS_THICKFRAME и WS_SYSMENU, WS_CAPTION, WS_MINIMIZEBOX, WS_MAXIMIZEBOX
WS_POPUP	создание временного окна
WS_ POPUPWINDOW	создание временного окна на базе стилей WS_BORDER, WS_POPUP и WS_SYSMENU
WS_SYSMENU	(только для окон с заголовком) создание окна с кнопкой вызова системного меню вместо стандартной кнопки, позволяющей закрыть окно при использовании этого стиля для дочернего окна
WS_TABSTOP	(только в диалоговых окнах) указывает на произвольное количество элементов управления (стиль WS_TABSTOP), между которыми можно перемещаться клавишей <TaЬ>
WS_THICKFRAME	создание окна с толстой рамкой, используемой для изменения размеров окна
WS_VISIBLE	создание окна, видимого сразу после создания
WS_VSCROLL	создание окна с вертикальной полосой прокрутки

Ниже приведен текст типового каркаса MFC-приложения с “заглушками” для обработчиков

```
//=================================================
//              ТИПОВОЙ КАРКАС MFC-ПРИЛОЖЕНИЯ (ТКП)
//=================================================
// 1. Создать мастером Win32 Application “пустой” каркас (тип empty).
// 2. Создать в каркасе пустой файл *.cpp.
// 3. Вставить текст ТКП в файл *.cpp.
```

```
// 4. Включить в главном меню в пункте Project-Settings, на вкладке General
//     режим – “use MFC in Static library”.
//=====================================================
#include        <afxwin.h>
//#include      <afxext.h>
//#include      <afxdisp.h>
//#include      <afxdtctl.h>
//#ifndef       _AFX_NO_AFXCMN_SUPPORT
   //#include   <afxcmn.h>
//#endif
#include        <iostream>
using namespace std;

// ===== класс главного ОКНА приложения ==================
class WINDOW : public CFrameWnd
{
public:
        WINDOW ();
      // afx_msg void  OnLButtonDown (UINT flags, CPoint loc);
      // afx_msg void  OnPaint ();
      // afx_msg void On<ИмяОбработчика>( < ПараметрыОбработчика > );
        DECLARE_MESSAGE_MAP()
};

// ===== конструктор Создание окна
WINDOW::WINDOW ()
{
      Create ( NULL, "Главное окно MFC-приложения” );
}

// ===== метод Обработчик сообщения
// afx_msg void WINDOW :: OnLButtonDown (UINT flags, CPoint  loc)
// { CClientDC  dc (this); < ФРАГМЕНТ_ПОЛЬЗОВАТЕЛЯ >  };

// ===== метод Обработчик сообщения
// afx_msg void WINDOW :: OnPaint ()
// { CPaintDC  dc (this); < ФРАГМЕНТ_ПОЛЬЗОВАТЕЛЯ >  };

// ===== метод Обработчик сообщения
//afx_msg void CMainWin::On< ИмяОбработчика >( < ПараметрыОбработчика > )
// { CClientDC  dc (this); < ФРАГМЕНТ_ПОЛЬЗОВАТЕЛЯ > };

// ===== очередь сообщений главного окна
BEGIN_MESSAGE_MAP(WINDOW, CFrameWnd)
    // ON_WM_PAINT()
    // ON_WM_LBUTTONDOWN()
    // ON_WM_<ИмяСообщения>( [ < Параметры > ] )
END_MESSAGE_MAP()

// ===== класс ПРИЛОЖЕНИЕ ==================
```

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

// ===== конструктор Инициализация приложения
BOOL APPLICATION::InitInstance ()
{
        m_pMainWnd = new  WINDOW;
        m_pMainWnd -> ShowWindow ( m_nCmdShow);
        m_pMainWnd -> UpdateWindow ();
        return TRUE;
}

// ===== создание экземпляра приложения ==================
APPLICATION    TheApplication;
```

Ниже приведен текст типового каркаса MFC-приложения с минимальным набором средств и без обработчиков сообщений

```
//=======================================================
//               ТИПОВОЙ КАРКАС MFC-ПРИЛОЖЕНИЯ (ТКП)
//=======================================================
#include    <afxwin.h>
#include    <iostream>
using namespace std;

class WINDOW : public CFrameWnd
{
public:
        WINDOW ();
        DECLARE_MESSAGE_MAP()
};
WINDOW::WINDOW ()
{
      Create ( NULL, "Главное окно типового каркаса MFC-приложения”);
}
BEGIN_MESSAGE_MAP(WINDOW, CFrameWnd)
END_MESSAGE_MAP()

class APPLICATION : public CWinApp
{
public:
      BOOL InitInstance();
};
BOOL APPLICATION::InitInstance ()
{
        m_pMainWnd =  new  WINDOW;
```

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

APPLICATION   TheApplication;   .
```

Лекция № 9 Обработка сообщений

1 Сообщения. Обработчики сообщений

“Общение” MFC-приложения и его окружения (внешней среды) производится через систему сообщений. Сообщения соответствуют происходящим событиям, связаны с конкретным источником (например, сообщения “мыши”, клавиатуры, диалогового окна, списка, кнопки и т.п.), предполагают реакцию приложения (обработку сообщений) путем запуска функций-обработчиков сообщений. Конкретный алгоритм обработки сообщений, как правило, определяется программистом.

В качестве внешней среды приложения выступают: – операционная система, через которую собственно реализуется обмен сообщениями; - пользователь, который оказывает воздействие на приложение (посылая сообщения) через устройства ввода (“мышь”, клавиатуру и т.п.) и к которому направлен поток информации (на устройства вывода – экран, принтер и т.п.), являющийся результатом обработки сообщений; - другие приложения, которые через ОС посылают-принимают сообщения (кроме этого, возможно и классическое взаимодействие приложений, например, через общую информацию, файлы, “почтовые ящики” и т.д.); - устройства, в том числе удаленные.

Основные, типовые объекты, средства, обеспечивающие общение пользователя с приложением, – это графические ресурсы приложения, образующие видимый (визуальный) интерфейс на экране дисплея. В Windows это окна (окна с клиентской областью, диалоговые, окна сообщений, модальные и немодальные окна, специализированные окна - элементы управления), в том числе, встраиваемые в приложение редактором ресурсов путем редактирования программистом соответствующих шаблонов, и встраиваемые программно как “самостоятельные” окна, создаваемые create. Реализация общения с целью ввода-вывода информации предполагает организацию в приложении связи (канала связи) ПРИЛОЖЕНИЕ-РАБОЧЕЕ_ОКНО.

Таким образом, приложение может как посылать сообщения (в том числе и самому себе), требуя определенной реакции от внешней среды (ОС, других приложений), так и принимать сообщения (через ОС от приложений, от самой ОС), реагируя соответствующим образом путем запуска функций-обработчиков сообщений. Это могут быть как готовые, типовые обработчики (например, обработчик OnOk() сообщения “нажатие кнопки - ОК” диалогового окна), которые программист может тем не менее переопределить. Так и “пустые” обработчики сообщений со стандартными интерфейсами, функционирование которых определяется программистом.

В библиотеке MFC для обработки сообщений применяются аналогичные используемым в Windows-приложениях (по назначению, по наименованию и схожие по сигнатуре) функции-обработчики, являющиеся членами классов MFC (например, класса CWnd). В них отсутствует первый параметр – дескриптор приложения HWND hwnd, могут присутствовать дополнительные уточняющие параметры. Типовой прототип такой afx-функции

afx_msg void ИМЯ_ОБРАБОТЧИКА (*ПараметрыСообщения*) .

Например, прототип обработчика события - нажатие клавиши клавиатуры (сообщение WM_CHAR), *ПараметрыСообщения* - ASCII-код нажатой клавиши, число повторных нажатий символа из-за удерживания клавиши, сопутствующие события (нажатые клавиши)

afx_msg void OnChar (UINT ch, UINT count,UINT flags)

Ниже приведен пример прототипа пользовательского обработчика события – нажатие кнопки диалогового окна класса MY_DIALOG (сообщение WM_COMMAND),

afx_msg void OnButtonOK() .

Сам обработчик описывается как

afx_msg void MY_DIALOG::OnButtonOK() {…}.

Прототипы некоторых функций-обработчиков сообщений библиотеки MFC приведены ниже в таблице 19.

Таблица 19 - Прототипы обработчиков сообщений

Сообщение	Прототип функции обработки сообщения
WM_CREATE	afx_msg int OnCreate(LPCREATESTRUCT *lpCreateStruct*); вызывается для извещения о завершении части этапа создания окна – от момента, когда окно создается функцией create, до момента возврата из функции и вывода окна на экран
WM_SHOWWINDOW	afx_msg void OnShowWindow(BOOL *bShow*, UINT *nStatus*); вызывается, когда окно (CWnd-объект) выводится или сворачивается методом ShowWindow, когда окно стиля overlapped максимально разворачивается, сворачивается в пиктограмму и т.п.
WM_SETFOCUS	afx_msg void OnSetFocus(CWnd* *pOldWnd*); вызывается при получении фокуса ввода
WM_KILLFOCUS	afx_msg void OnKillFocus(CWnd* *pNewWnd*); вызывается сразу перед потерей фокуса ввода
WM_SIZE	afx_msg void OnSize(UINT *nType*, int *cx*, int *cy*); вызывается после того, как размеры окна были изменены
WM_SIZING	afx_msg void OnSizing(UINT *nSide*, LPRECT *lpRect*); вызывается, извещая, что пользователь меняет размеры окна
WM_MOVE	afx_msg void OnMove(int x, int y); вызывается, когда пользователь переместил окно (CWnd-объект)
WM_MOVING	afx_msg void OnMoving(UINT *nSide*, LPRECT *lpRect*); вызывается, пока пользователь перемещает окно (CWnd-объект)
WM_TIMER	afx_msg void OnTimer(UINT nIDEvent); вызывается по истечении каждого интервала времени, специфицированного в методе установки таймера SetTimer

WM_HSCROLL
WM_VSCROLL
	afx_msg void OnHScroll(UINT *nSBCode*, UINT *nPos*, CScrollBar* *pScrollBar*);
	afx_msg void OnVScroll(UINT *nSBCode*, UINT *nPos*, CScrollBar* *pScrollBar*);
	вызывается, когда пользователь совершает щелчок по горизонтальной или вертикальной полосе прокрутки окна

WM_PAINT	afx_msg void OnPaint( ); вызывается, когда Windows или приложение посылает запрос на перерисовку окна (методы UpdateWindow, RedrawWindow)
WM_CLOSE	afx_msg void OnClose( ); система вызывает этот обработчик как сигнал, что окно (CWnd-объект) или приложение нужно завершить. Обработка по умолчанию вызывает DestroyWindow
WM_DESTROY	afx_msg void OnDestroy( ); система вызывает этот обработчик, чтобы оповестить окно (CWnd-объект), что оно разрушается (OnDestroy вызывается после того, как окно убирается с экрана)
WM_CHAR	afx_msg void OnChar(UINT *nChar*, UINT *nRepCnt*, UINT *nFlags*); вызывается нажатием несистемной клавиши (до метода OnKeyUp и после OnKeyDown). Сообщает значение нажатой клавиши
WM_KEYDOWN	afx_msg void OnKeyDown(UINT *nChar*, UINT *nRepCnt*, UINT *nFlags*); вызывается нажатием несистемной клавиши
WM_KEYUP	afx_msg void OnKeyUp(UINT *nChar*, UINT *nRepCnt*, UINT *nFlags*); вызывается отжатием несистемной клавиши
WM_COMMAND	virtual BOOL OnCommand(WPARAM *wParam*, LPARAM *lParam*); вызывается, когда пользователь выбирает команду (выбирает пункт меню, нажимает кнопку и т.д.)
WM_RBUTTONDOWN, WM_LBUTTONDOWN	afx_msg void OnLButtonDown(UINT *nFlags*, CPoint *point*); вызывается, когда пользователь нажимает левую или правую кнопку “мыши”
WM_LBUTTONUP WM_RBUTTONUP	afx_msg void OnLButtonUp(UINT *nFlags*, CPoint *point*); вызывается, когда пользователь отжимает левую или правую кнопку “мыши”
WM_LBUTTONDBLCLK WM_RBUTTONDBLCLK	afx_msg void OnLButtonDblClk(UINT *nFlags*, CPoint *point*); вызывается двойным щелчком левой или правой кнопкой “мыши”
WM_MOUSEMOVE	afx_msg void OnMouseMove(UINT *nFlags*, CPoint *point*); вызывается, когда перемещается курсор “мыши”

1. Сообщение WM_CHAR вызывается каждый раз после нажатия несистемной клавиши (перед методом OnKeyUp и после метода OnKeyDown), содержит код клавиши (нажатой или отжатой) и обрабатывается функцией-обработчиком с прототипом

afx_msg void OnChar(UINT *ASCII_КодКлавиши*, UINT *ЧислоПовторов*, UINT *ОписаниеСитуации*) ,

где параметр *ЧислоПовторов* – указывает число повторных нажатий символа из-за удерживания клавиши;

параметр *ОписаниеСитуации* (флаги) содержит скэн-код клавиши, предыдущее состояние клавиши, описывает сопутствующие события (нажатые клавиши) (см. таблицу 20 ниже).

Таблица 20 - Значения флагов

Флаг	Значение
16-23	скэн-код клавиши, зависящий от конкретной аппаратуры
24	1, если клавиша является функциональной (на расширенной клавиатуре, например, правосторонние клавиши ALT, CTRL)
29	1, если была нажата и клавиша ALT (иначе 0)
30	1, если клавиша нажата до посылки сообщения (описывает предшествующее состояние клавиатуры)
31	1, если клавиша отпускается; 0, если удерживается нажатой

2. Сообщение WM_COMMAND вызывается каждый раз, когда пользователь выбирает элемент управления (например, “мышью” выбирает конечный пункт меню, нажимает кнопку и т.п.). Для включения реакции приложения на происшедшее событие используется макрокоманда ON_COMMAND(*ID_ЭлементаУправления*, *ИмяОбработчика*) ,где параметр *ID_ЭлементаУправления* задает идентификатор выбранного ресурса интерфейса (например, пункта меню, кнопки и т.п.), а *ИмяОбработчика* определяет вызываемую функцию.

3. Сообщение WM_DESTROY вызывается каждый раз, когда CWnd-объект удаляется с экрана, и обрабатывается функцией-обработчиком с прототипом afx_msg void OnDestroy( ).

4. Сообщение типа WM_KILLFOCUS вызывается каждый раз - сразу же перед потерей фокуса ввода (the input focus) и обрабатывается функцией-обработчиком с прототипом afx_msg void OnKillFocus(CWnd * *УказательНовогоОкна*) ,где параметр сообщает *УказательНовогоОкна*, которое принимает фокуса ввода (может быть NULL).

5. Сообщение типа WM_LBUTTONDOWN вызывается каждый раз после нажатия соответствующей клавиши “мыши” и обрабатывается функцией-обработчиком с прототипом afx_msg void OnLButtonDown(UINT *ОписаниеСитуации*, CPoint *Координаты Курсора*), где параметр *КоординатыКурсора* – определяет *КоординатуКурсора.x* и *КоординатуКурсора.y* в момент нажатия *клавиши. А параметр ОписаниеСитуации (флаги),* описывает, была ли попутно нажата и удерживалась клавиша клавиатуры или “мыши” (например, MK_CONTROL - была нажата клавиша CTRL; MK_SHIFT - нажата клавиша SHIFT; MK_LBUTTON, MK_MBUTTON, MK_RBUTTON - удерживалась соответственно левая, средняя или правая кнопка “мыши”).

Аналогичные прототипы у обработчиков сообщений типа WM_RBUTTONDOWN, WM_LBUTTONDBLCLK, WM_RBUTTONDBLCLK - afx_msg void OnRButtonDown(…), afx_msg void OnLButtonDblClk(…), afx_msg void OnRButtonDblClk(…). Последние два сообщения принимают только окна со стилем CS_DBLCLKS (заданным в параметре style структуры WNDCLASS).

6. Сообщение WM_TIMER вызывается каждый раз после истечения каждого интервала времени, указанного в методе SetTimer, используемом для установки таймера, и обрабатывается функцией-обработчиком с прототипом afx_msg void OnTimer(UINT *ИдентификаторТаймера*) ,где параметр идентифицирует сработавший таймер.

7. Сообщение WM_PAINT вызывается каждый раз после выполнения методов UpdateWindow, RedrawWindow, вызывается каждый раз, когда требуется перерисовка-обновление рабочей, клиентской области экрана и т.п. и обрабатывается функцией-обработчиком с прототипом afx_msg void OnPaint( ) .

8. Сообщение WM_SIZE вызывается каждый раз, после того как изменились размеры окна приложения, и обрабатывается функцией-обработчиком с прототипом afx_msg void OnSize(UINT *ОписаниеСитуации*, int *Ширина*, int *Высота*) ,где параметр *ОписаниеСитуации* указывает причину сообщения (например, SIZE_MAXIMIZED, SIZE_MINIMIZED - окно было максимизировано или минимизировано, SIZE_RESTORED - размеры окна произвольно изменил пользователь и т.д.); параметры *Ширина* и *Высота* дают новые размеры клиентской области окна.

2 Организация ввода-вывода

Контекст устройства. Для управления устройствами (например, принтером, дисплеем) в операционной системе Windows реализована концепция аппаратно-независимого программирования без прямого доступа к устройствам. Доступ осуществляется через специальный однотипный (по функциональному составу) интерфейс, реализуемый функциями API, при этом конкретные устройства представляются их виртуальными, графическими аналогами. В библиотеке MFC указанное поддерживается с помощью атрибутов класса CDC. Класс содержит целый ряд свойств, называемых атрибутами контекста устройства (содержащих данные о графическом устройстве вывода, задающими, например, цвет линии и т.п.). Кроме этого, класс CDC содержит все методы для построения изображения в окне.

Для осуществления в приложении операций по вводу-выводу информации необходимо организовать связь (канал связи) ПРИЛОЖЕНИЕ—РАБОЧЕЕ_ОКНО (также как при проведении файловых операций ввода-вывода необходимо открыть канал связи ПРИЛОЖЕНИЕ—ФАЙЛ). По терминологии Windows это означает, что приложению требуется контекст устройства (КУ). С информационной точки зрения контекст устройства - специальная служебная структура, которая предварительно создается, инициализируется (заполняется). Для этого используется класс CDC и производные классы CClientDC, CPaintDC, определенные в библиотеке MFC, а при реализации ввода-вывода применяются их методы.

Получение контекста устройства производится путем создания экземпляра класса CClientDC либо экземпляра класса CPaintDC. В первом случае используется конструктор CClientDC (*УказательНаРабочееОкноПриложения*) или CClientDC (CWnd *Window). Здесь роль указателя на класс CWnd - РабочееОкно обычно выполняет указатель по умолчанию this (т.е. *ЭтоРабочееОкно*). Например, команда CClientDC TheDC (this) позволяет создать (получить) КУ как экземпляр TheDC класса CClientDC, инициализированный значением this.

Во втором случае, который применяется при получении КУ в обработчике сообщения WM_PAINT, используется конструктор CPaintDC (*УказательНаРабочееОкноПриложения*) или CPaintDC TheDC (CWnd *Window).Например,

```
OnPaint()
{
    CPaintDC  TheDC (this);
…
}  .
```

Перерисовка. При работе с интерфейсным элементом типа окно возникают проблемы перерисовки (обновления) окна. Они связанны как с потерей и соответственно необходимостью восстановления облика окна, которое было свернуто, закрыто, перекрыто и т.п. в ходе работы приложения так и с необходимостью обновления облика окна при изменении выводимой информации. Говорят, что изображение в окне может

быть “повреждено” - на время перекрыто другим окном. В Windows нет автоматического сохранения текущего облика с последующим автоматическим восстановлением окна, нет автоматического перерисовки. За исключением ряда специфических манипуляций, когда оно сохраняется и восстанавливается в прежнем виде – например, при перемещении окна, масштабировании. Если область перекрытия незначительная, например, окном меню, то перекрытая часть восстанавливается системой Windows. Это выполнено сознательно, чтобы не работать с избыточной информацией и не замедлять обслуживание параллельно исполняемых приложений. Решение указанных проблем возложено на программиста (для восстановления при необходимости облика окна он может сам сохранять нужные данные либо вычислять их по известным ему протоколам или алгоритмам работы приложения, использовать средства MFC по организации “виртуальных окон” и т.д.). В большинстве же случаев Windows лишь сообщает приложению посылкой сообщения о необходимости восстановить окно. Для этого система Windows фиксирует информацию о поврежденной области (invalid region) или о поврежденный прямоугольнике области усечения (invalid rectangle), т.е. минимальном прямоугольнике, покрывающем поврежденную область. Описание поврежденного прямоугольника и поврежденной области можно получить методами CWnd ::GetUpdateRect и CWnd ::GetUpdateRgn.

Для восстановления поврежденного окна используются обработчики типа OnPaint, OnDraw. Для запуска этих обработчиков можно генерировать программно соответствующие сообщения типа WM_PAINT, передавая указатели на контекст устройства, используя который, можно получить параметры поврежденной области или прямоугольника. Для этого используются методы CWnd ::Invalidate, CWnd ::InvalidateRect и CWnd ::InvalidateRgn, которые позволяют объявить соответственно клиентскую область, некоторый прямоугольник и область окна поврежденными и послать сообщение типа WM_PAINT в очередь приложения. Методы CWnd::ValidateRect и CWnd ::ValidateRgn позволяют отменить объявление некоторого прямоугольника или области поврежденными, что ведет к корректировке текущего поврежденного прямоугольника.

Метод InvalidateRect позволяет объявить часть клиентской области (в указанном прямоугольнике) “недействительной”, “поврежденной”, требующей перерисовки и добавить ее к текущей области, требующей обновления. Это вынуждает Windows послать приложению сообщение WM_PAINT. Указанная область вместе с другими, требующими обновления, может быть перерисована по получении нового сообщения WM_PAINT. Метод имеет прототип

ПЕРЕРИСОВАТЬ_ОКНО (LPCRECT *УказательОбластиПерерисовкиВнутриОкна*, BOOL *СтеретьФонОкна* = TRUE) или void CWnd:: InvalidateRect (LPCRECT *lpRect*, BOOL *bErase* = *TRUE*) ,где параметр lpRect – указатель на CRect-объект или RECT-структуру, которая идентифицирует прямоугольник, требующий перерисовки и добавляемый к области обновления (если NULL, то обновляется вся клиентская область); - bErase – флаг перерисовки, который определяет необходимость перерисовки фона области (при этом прежнее изображение стирается). Если новое изображение перекрывает старое, то фон можно не стирать.

ПРИМЕР использования метода представлен ниже. Здесь прямоугольные области, описанные переменными rctRect1 и rctRect2, требуют перерисовки. Причем фон первой области стирается, а второй – нет.

```
RECT rctRect1;
rctA.left = 20;
```

```
rctA.top = 20;
rctA.right  = 80;
rctA.bottom = 80;
RECT rctRect2= {100,100,300,400};
InvalidateRect ( LPCRECT rctRect1);
InvalidateRect ( LPCRECT rctRect2, FALSE );   .
```

Метод Invalidate объявляет всю клиентскую область CWnd-объекта “недействительной” и имеет прототип void CWnd:: Invalidate(BOOL *bErase* = TRUE). Метод InvalidateRgn позволяет объявить указанную область (часть клиентского окна) “недействительной”, “поврежденной”, требующей перерисовки, и имеет прототип void CWnd:: InvalidateRgn(CRgn* *pRgn*, BOOL *bErase* = TRUE) ,где параметр pRgn – указатель на CRgn-объект, который идентифицирует область (регион), добавляемый к области обновления (если NULL, то это вся клиентская область).

Методы ввода-вывода. В общем случае для организации вывода информации, например, в рабочую область окна приложения, следует предварительно получить контекст устройства КУ и соответственно получить доступ к необходимым функциям КУ.

1. ВЫВОД информации (числовых и не числовых данных) в виде строк в специальном окне - окне сообщений. Для этих целей можно использовать метод MessageBox() класса CWnd либо глобальную функцию afxMessageBox().

Прототип метода CWnd::MessageBox - int CWnd:: MessageBox (LPCSTR *lpszText*, LPCSTR *lpszTitle* = NULL, UINT *MBType* = MB_OK) или ВЫВЕСТИ_ОКНО_СООБЩЕНИЯ (*СтрокаСообщения*, *НазваниеОкнаСообщения*, *СоставЭлементовОкнаСообщения*).

Метод создает и выводит окно с заголовком, сообщением, комбинацией пиктограмм и командных кнопок. Возвращает значение, определяющее результат работы пользователя с окном - код нажатой кнопки (или 0, если недостаточно памяти для создания окна). Коды завершения приведены в таблице 21 ниже

Таблица 21 - Коды завершения

Значение	Описание
IDABORT	выбрана кнопка Abort
IDCANCEL	выбрана кнопка Cancel
IDIGNORE	выбрана кнопка Ignore
IDNO	Ни одна кнопка не была выбрана
IDOK	выбрана кнопка OK
IDRETRY	выбрана кнопка Retry
IDYES	выбрана кнопка Yes

Параметры:

- *СтрокаСообщения* - указатель на CString-объект или строку C (с нуль-символом), которая содержит выводимое сообщение;
- *НазваниеОкнаСообщения* - указатель на CString-объект или строку C (с нуль-символом), которая содержит заголовок окна сообщения. Если указатель равен NULL, то по умолчанию используется заголовок “Error”;
- *СоставЭлементовОкнаСообщения* - определяет состав окна сообщения (комбинацию пиктограмм и командных кнопок) и его поведение. Используемые пиктограммы: MB_ICONHAND, MB_ICONSTOP, MB_ICONERROR, MB_ICONQUESTION, MB_ICONEXCLAMATION, MB_ICONWARNING, MB_ICON-

ASTERISK, MB_ICONINFORMATION. Используемые комбинации кнопок: MB_ABORTRETRYIGNORE, MB_OK, MB_YESNO, MB_OKCANCEL, MB_RETRYCANCEL, MB_YESNOCANCEL.

Установки модальности окна: MB_APPLMODAL (используется по умолчанию. В этом случае пользователь для продолжения работы в текущем окне должен ответить на окно сообщения. Тем не менее, он может перейти в окна других приложений и продолжить работу там), MB_SYSTEMMODAL (применяется для оповещения пользователя о серьезных ситуациях, не терпящих отлагательства).

Установки кнопок по умолчанию: MB_DEFBUTTON1 (первая кнопка), MB_DEFBUTTON2 (вторая кнопка), MB_DEFBUTTON3 (третья кнопка).

Пример использования функции - MessageBox (“Работает MessageBox”, "Демо", MB_OK); .

Пример использования окна сообщения для ввода управляющей информации в виде данных о реакции пользователя (нажатых кнопках), представлен ниже

```
if  ( MessageBox ("Работает MessageBox ", "Демо", MB_OKCANCEL) == IDCANCEL)
      MessageBox ("Нажата кнопка CANCEL", "Демо", MB_OKCANCEL);  .
```

Прототип функции afxMessageBox, которая функционально работает так же как и предыдущий метод, но реализована как глобальная функция MFC, приведен ниже

int afxMessageBox(LPCSTR *lpszText*, UINT *MBType* = MB_OK, UINT *HelpID* = 0)

или ВЫВЕСТИ_ОКНО_СООБЩЕНИЯ (*СтрокаСообщения*, *СоставЭлементовОкнаСообщения*, *УказательРазделаСправки*).

Другая используемая форма

int AFXAPI AfxMessageBox(UINT *nIDPrompt*, UINT *nType* = MB_OK, UINT *nIDHelp* = (UINT) -1),

где - *lpszText* указатель на CString-объект или строку С (с нуль-символом), которая содержит выводимое сообщение;

- *nType* определяет стиль окна - состав окна сообщения и его поведение;

- *nIDHelp* определяет ID–идентификатор контекстно-зависимого справочного сообщения, связанного с этим окном;

- *nIDPrompt* - определяет ID–идентификатор строки в таблице строк ресурсов приложения, которая используется в качестве выводимого сообщения.

2. ВЫВОД информации (числовых и не числовых данных) в виде строк в клиентской области окна. Требует предварительного получения КУ. Сам вывод можно организовать методом TextOut, который перегружен в двух вариантах – для работы с классическими строками С и со string-строками. Прототип метода CDC::TextOut

virtual BOOL CDC:: TextOut (int *X*, int *Y*, LPCSTR *lpszString*, int *Length*) ,
BOOL CDC:: TextOut (int *X*, int *Y*, const CString &*StrOb*)

или ВЫВЕСТИ_ТЕКСТ (*КоординатаX*, *КоординатаY*, *СтрокаВывода*, *ДлинаСтрокиВывода*) и соответственно ВЫВЕСТИ_ТЕКСТ (*КоординатаX*, *КоординатаY*, *СтрокаВывода*).

Пример использования метода

```
CClientDC  dc(this);
char  OutputString;
char  Comment [25] = “Выводится строка - ”;
```

```
……
dc.TextOut(1,1, Comment, strlen(Comment));
dc.TextOut(1,20, OutputString, strlen(OutputString));
```

В примере, приведенном ниже, в приложении, созданном на базе ТКП, при нажатии левой кнопки “мыши” (сообщение WM_LBUTTONDOWN и обработчик OnLButtonDown) осуществляется вывод заранее инициализированной строки OutputStr в клиентскую область экрана, начиная с позиции, в которой было зафиксировано положение курсора “мыши” в момент события.

```
char OutputStr[80] = " Строка вывода ";

class CMainWin : public CFrameWnd
{
public:
     ……
     afx_msg void OnLButtonDown(UINT flags, CPoint loc);
     DECLARE_MESSAGE_MAP()
};

CMainWin::CMainWin()
{
     Create(NULL, "Пример");
}

BEGIN_MESSAGE_MAP(CMainWin, CFrameWnd)
  ON_WM_LBUTTONDOWN()
END_MESSAGE_MAP()

afx_msg void CMainWin::OnLButtonDown(UINT flags, CPoint loc)
{
     CClientDC dc(this);
      dc.TextOut(loc.x,loc.y, OutputStr, strlen(OutputStr));
};

class CApp : public CWinApp
{
public:
     BOOL InitInstance();
};

BOOL CApp::InitInstance()
{
     m_pMainWnd = new CMainWin;
     m_pMainWnd->ShowWindow(m_nCmdShow);
     m_pMainWnd->UpdateWindow();
     return TRUE;
}

CApp  App;  .
```

3. ВЫВОД информации (числовых и не числовых данных) может производиться в виде строк в стандартных элементах управления (ЭУ), например, в окошке редактирования (класс CEdit), в списковом окне (класс CList) и т.п.

4. ВВОД ограниченной информации может производиться в виде получения реакции, ответа пользователя на полученное в окне сообщение. Вводимая информация представляет собой код нажатой пользователем кнопки окна сообщения, выведенного функцией MessageBox.

5. ВВОД произвольных данных в строковом формате может производиться из окна (поля) редактирования ресурса – диалоговое окно.

Лекция № 10 Разработка интерфейсов на базе библиотеки MFC

1 Графический интерфейс приложения. Диалоговые окна

Графический интерфейс приложения предназначен обеспечивать интерактивный доступ к приложению со стороны пользователей. В ОС Windows для этого используются специальные графические объекты.

Как правило, это окна различных типов, включая диалоговые окна, служащие подложкой, где могут располагаться различные элементы управления (ЭУ) и сами элементы управления.

Как при создании Windows-приложений, так и их графических интерфейсов широко используются классы библиотеки MFC, что ускоряет и упрощает разработку приложений. Фрагмент иерархии классов библиотеки MFC, используемых для создания приложений с графическим интерфейсом, представлен ниже (рисунки 36, 37).

Классы для работы с ДО. С типовым MFC-приложением связывается определяющий его на верхнем уровне объект, принадлежащий классу, производному от класса CWinApp.

Фрагменты иерархии классов библиотеки MFC, используемых для управления диалоговыми окнами и элементами управления представлены на рисунках ниже.

Оконный интерфейс приложения строится как система взаимодействующих окон различных стилей и типов, производных от класса CWnd.

Так для приложений с однодокументной архитектурой интерфейс строится на базе классов CFrameWnd, CDialog, семейства классов элементов управления (ЭУ) и др. Диалоговые окна интерфейса, являющиеся подложкой для размещения и компоновки различных ЭУ, производятся от класса CDialog (рисунок 38).

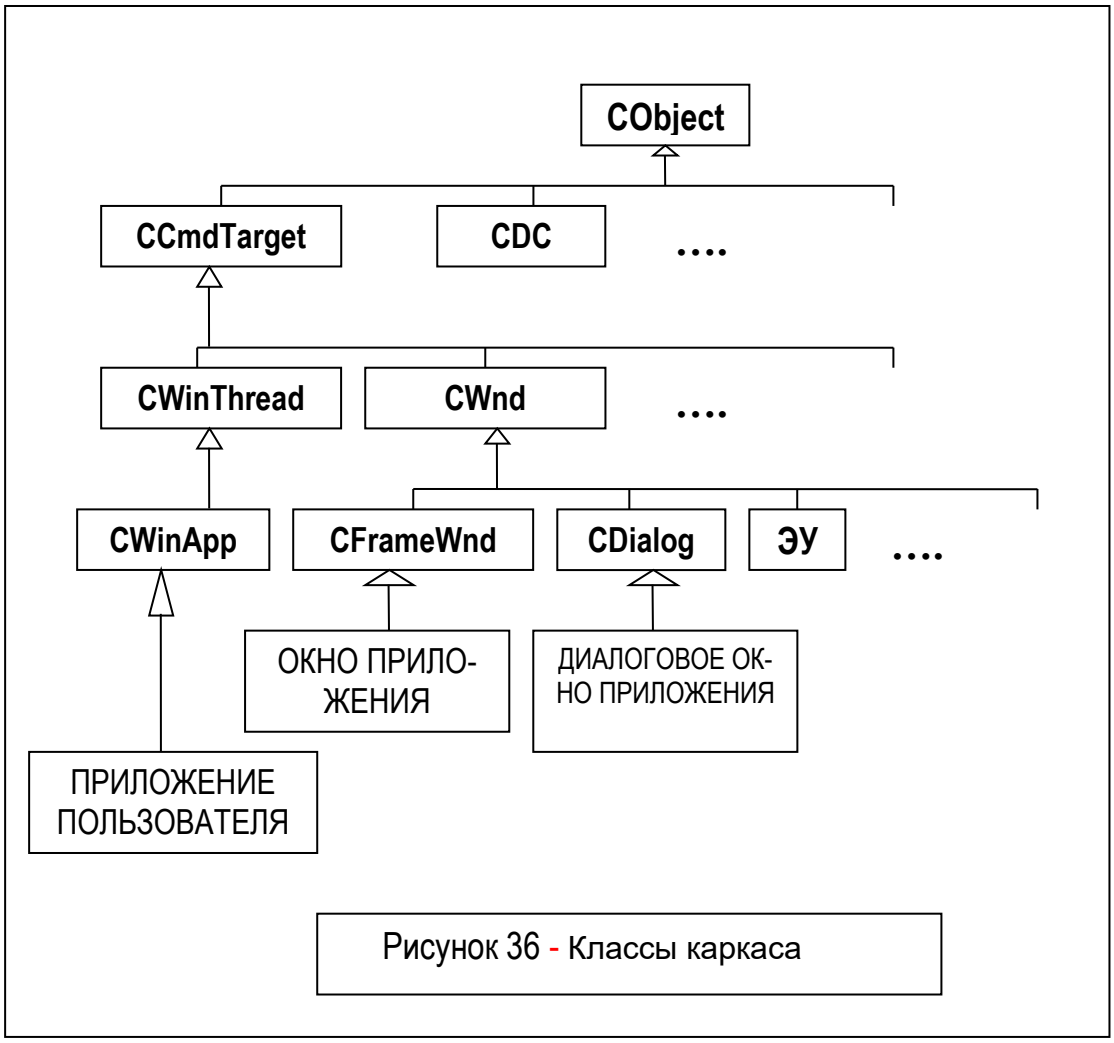


Рисунок 36 - Классы каркаса

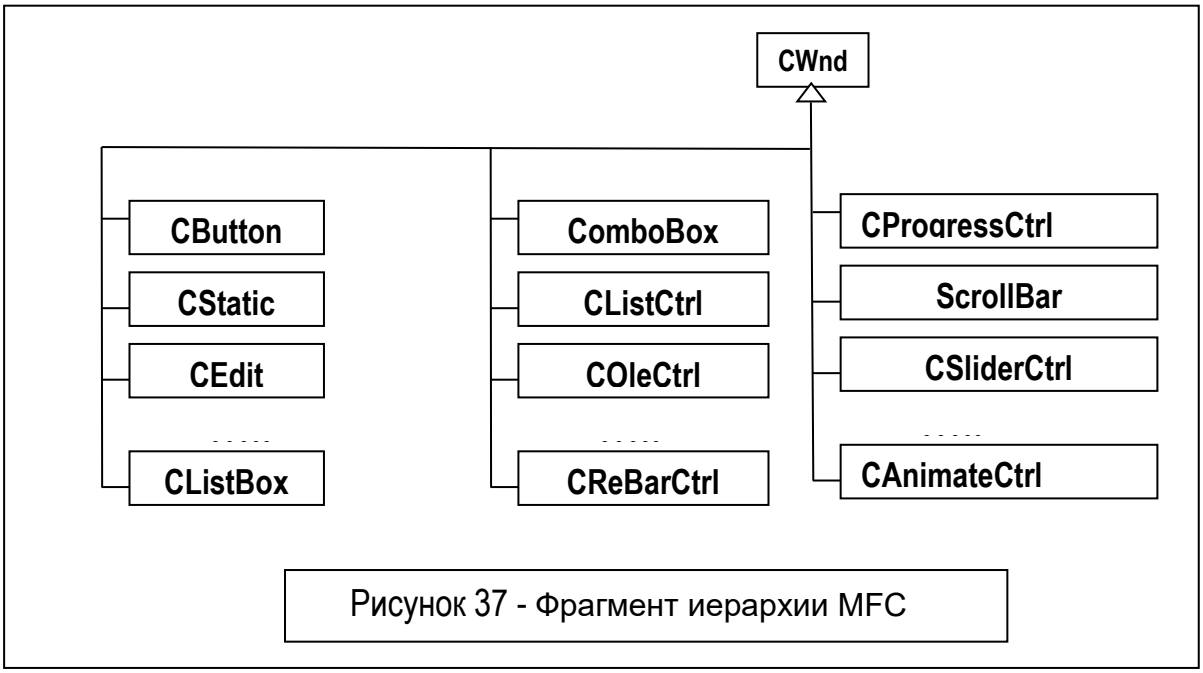


Рисунок 37 - Фрагмент иерархии MFC

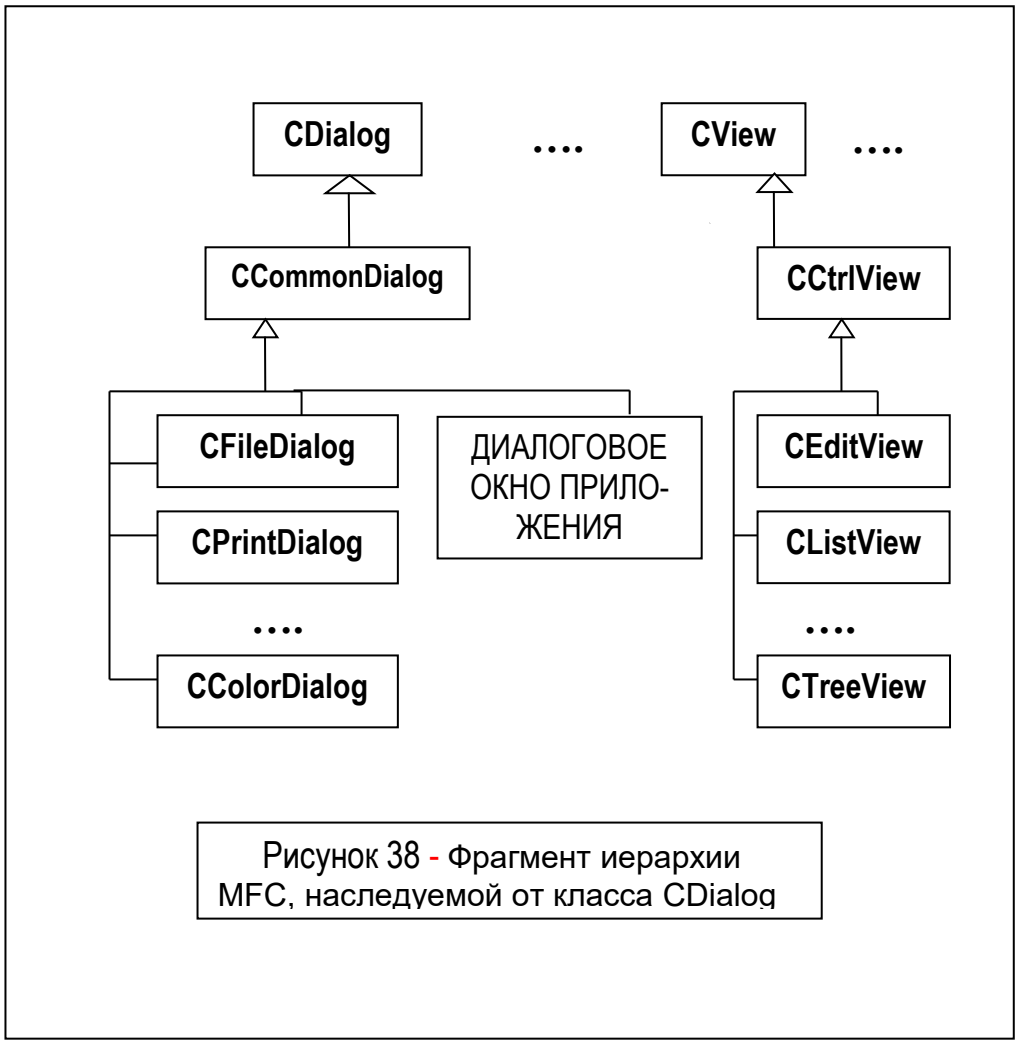


Рисунок 38 - Фрагмент иерархии MFC, наследуемой от класса CDialog

Каждый ЭУ, представляющий собой специфическое окно, производится от соответствующего класса семейства классов элементов управления. Стили окон и диалоговых окон приведены в таблицах 22, 23 ниже.

Таблица 22 - Стили окон и диалоговых окон

Параметр стиля	Описание
WS_BORDER	создание окна с рамкой
WS_CAPTION	добавление к окну с рамкой заголовка
WS_CHILD	создание дочернего окна
WS_CHILDWINDOW	создание дочернего окна стиля WS_CHILD
WS_CLIPCHILDREN	при создании родительского окна запрещает его рисование в области, занятой любым дочерним окном
WS_CLIPSIBLINGS	(только со стилем WS_CHILD) не использовать при получении данным окном сообщения о перерисовке области, занятые другими дочерними окнами. Иначе разрешается рисовать в клиентской области другого дочернего окна
WS_DISABLED	создание первоначально неактивного окна
WS_DLGFRAME	создание окна с двойной рамкой без заголовка
WS_GROUP	(только в диалоговых окнах) стиль указывается для первого элемента управления в группе элементов, пользователь

	перемещается между ними с помощью клавиш-стрелок
WS_HSCROLL	создание окна с горизонтальной полосой прокрутки
WS_MAXIMIZE WS_MINIMIZE	создание окна максимального (минимального) размера
WS_MAXIMIZEBOX WS_MINIMIZEBOX	создание окна с кнопкой максимизации (минимизации)
WS_OVERLAPPED	создание перекрывающегося окна
WS_OVERLAPPED WINDOW	создание перекрывающегося окна на базе стилей WS_OVERLAPPED, WS_THICKFRAME и WS_SYSMENU
WS_POPUP	(не используется с окнами стиля WS_CHILD) создание временного окна
WS_ POPUPWINDOW	создание временного окна на базе стилей WS_BORDER, WS_POPUP и WS_SYSMENU
WS_SYSMENU	(только для окон с заголовком) создание окна с кнопкой вызова системного меню вместо стандартной кнопки, позволяющей закрыть окно при использовании этого стиля для дочернего окна
WS_TABSTOP	(только в диалоговых окнах) указывает на произвольное количество элементов управления (стиль WS_TABSTOP), между которыми можно перемещаться клавишей <TaЬ>
WS_THICKFRAME	создание окна с толстой рамкой, используемой для изменения размеров окна
WS_VISIBLE	создание окна, видимого сразу после создания
WS_VSCROLL	создание окна с вертикальной полосой прокрутки

Таблица 23 - Стили диалоговых окон

Параметр стиля	Описание
DS_3DLOOK	использование трехмерных рамок для изображения элементов управления диалогового окна
DS_ABSALIGN	в качестве координат диалогового окна вместо клиентских используются экранные
DS_CENTER	центрирование диалогового окна в рабочей области родительского окна. Иначе центрирование производится в рабочей области монитора в соответствии с системными настройками
DS_CENTERMOUSE	центрирование диалогового окна относительно курсора “мыши”
DS_CONTEXTHELP	включает значок вопроса в поле заголовка диалогового окна
DS_FIXEDSYS	вызывает использование в диалоговом окне SYSTEM_FIXED_FONT вместо SYSTEM_FONT (по умолчанию)
DS_MODALFRAME	создает диалоговое окно с модальной рамкой. Можно комбинировать с полем названия и системным меню, задаваемыми стилями WS_CAPTION и WS_SYSMENU

Диалоговые окна. Основу графических интерфейсов образуют диалоговые окна, называемые так именно потому, что они позволяют пользователям “общаться” с приложением. Производится это общение посредством использования элементов управления,

размещенных в диалоговых окнах. Каждое окно, созданное программно в памяти компьютера либо на базе шаблона – ресурса (визуального или текстового описания), кроме внешнего облика, который видит пользователь, является объектом соответствующего библиотечного класса. При этом объект инициализируется параметрами окна (координатами, размером, стилем и т.д.), заданными при его описании, создании. Методы соответствующего класса используются программистом для управления поведением окна.

Элементы управления. Описание элемента управления (ЭУ) включает следующий набор параметров: а) назначение, реализуемые функции, пользовательский интерфейс. Описание диаграммы прецедентов; б) типы (виды) ЭУ. Стили ЭУ. Режимы использования; в) представление ЭУ на языке описания ресурсов (ресурсное описание); г) базовый класс (классы), используемые методы и данные. Описание диаграммы классов; д) собственный класс, используемые методы и данные. Описание диаграммы классов; е) методы, команды, меняющие состояние ЭУ; ж) генерируемые, посылаемые сообщения (типы, причины возникновения – события, параметры, обработчики); з) создание и разрушение ЭУ; и) описание состояний, диаграммы состояний ЭУ; к) особенности программной реализации, управления ЭУ.

Видимые ЭУ сами, как правило, представляют собой окна, специализированные по назначению, поддерживаемым функциям. Например, кнопки предназначены для фиксации события – нажатие, окна редактирования предназначены как для ввода так и отображения информации и функционально представляют собой упрощенный вариант текстового редактора и т.д. Соответственно ЭУ являются объектами библиотечных классов либо пользовательских классов, производных от библиотечных.

Пользователь может выполнять действия над ЭУ (например, вводить число в окне редактирования, выбирать “мышью” кнопку, одинарным или двойным щелчком выбирать строку в списке и т.д.). Все это события (ввода), приводящие к посылке с помощью ОС соответствующих сообщений.

Приложения, обработчики сообщений приложений, могут, используя методы соответствующих классов, поддерживающих ЭУ, менять их состояния, тем самым выводить информацию пользователю. Например, добавлять новую строку в окно-список, выводить число в окне редактирования и т.д.

Справочная информация. Для получения справочной информации по классам MFC, информации о типах и особенностях сообщений, а также о прототипах их обработчиков можно использовать справочную систему MSDN.

Например, для получения информации о классе CDialog можно выделить текст – “Cdialog” в листинге приложения (либо набрать его в качестве шаблона поиска в справочнике) и нажать клавишу F1. Будет выведено окно с перечнем найденных разделов, содержащих запрошенную информацию. Например, для CDialog, это следующие разделы: CDialog - общие сведения о базовых классах, назначении, поведении, способах создания диалогового окна; раздел CDialog Class Members – общие сведения о методах класса; раздел CDialog::CDialog – общие сведения о конструкторах класса и т. д.

Для поиска информации можно использовать указатель MSDN. Например, можно набрать слово “CEdit” в качестве ключевого слова для поиска информации.

Поиск информации об ЭУ (например, CEdit, CListBox и т.д.) может быть произведен также по слову “controls”. Среди найденных разделов следует выбрать – “Controls”. При этом в соответствующем подразделе можно найти информацию о сообщениях, посылаемых от элемента управления к операционной системе в результате воздействий пользователя и макрокомандах включения чувствительности к ним.

Может быть получена информация по конкретному сообщению - поиском по названию сообщения (например, WM_COMMAND или WM и т.д.).

Сообщения. Необходимость ввода-вывода информации предполагает организацию в приложении канала связи ПРИЛОЖЕНИЕ-ОКНО. При этом приложение может как посылать сообщения, требуя определенной реакции от внешней среды (ОС, других приложений), так и принимать сообщения, реагируя соответствующим образом путем запуска функций-обработчиков сообщений. Для этого могут использоваться как готовые, типовые обработчики так и “пустые” обработчики сообщений со стандартными интерфейсами, функционирование которых доопределяется программистом.

Во взаимодействии пользователя с приложением участвуют: 1) сами пользователи, которые воздействуют на элементы управления, совершают события и тем самым инициируют сообщения; 2) элементы управления, идентифицируемые, например, дескриптором ресурса, указателем соответствующего объекта, которые имеют графический интерфейс, воспринимающий действия пользователя. ЭУ характеризуются набором посылаемых сообщений; 3) окно-родитетель, подложка, где располагаются ЭУ и куда направляются сообщения на обработку. Окна должны быть чувствительно к сообщениям и должны иметь соответствующие обработчики.

При работе с ЭУ следует учитывать: 1) сообщения, генерируемые самим ЭУ, которые посылаются в ОС через параметры сообщения WM_COMMAND; 2) сообщения, адресуемые ЭУ от функций приложения, которые посылаются принудительно, например, с помощью функции SendMessage; 3) сообщения ОС. Например, WM_CUT и др. Они могут посылаться функцией SendMessage.

1. Сообщения первой группы (notification message) вызываются пользователем, который выполнил действия, могущие привести к изменению состояния ЭУ (например, текста в окне редактирования). Сообщение (например, EN_CHANGE для списка) генерируется ЭУ и посылается в ОС. При этом родительское окно получает его (код и параметры сообщения) через сообщение WM_COMMAND. Например, в обработчик поступает следующая информация

```
LRESULT CALLBACK Обработчик (
  HWND ДескрипторОкнаРодителя,
  UINT КодПолученногоСообщения,
  WPARAM АтрибутыПолученногоСообщения,
  LPARAM ДескрипторОкнаИсточникаСообщения
);  .
```

Здесь *КодПолученногоСообщения* - WM_COMMAND, младшее слово параметра *АтрибутыПолученногоСообщения* содержит идентификатор окна, а старшее слово специфицирует сообщение (например, как EN_CHANGE).

2. Сообщения второй группы позволяют программно влиять на состояние ЭУ, используя его методы. Для этого применяется функция SendMessage. Например, программист может послать сообщение EM_GETLINE окну редактирования для копирования указанной строки текста из этого окна в буфер

```
SendMessage (
  (HWND) ДескрипторОкнаНазначения,
  EM_GETLINE,
  (WPARAM) НомерЧитаемойСтроки,
  (LPARAM) УказательПриемникаСтроки          // тип LPCTSTR
);   .
```

Для того, чтобы приложение реагировало на сообщение о происшедшем событии, необходимо: 1. Включить чувствительность приложения к сообщению, связав источник сообщения и функцию-обработчик сообщения. Указанное выполняется макрокомандой включения вида ON_*Сообщение* (*ДескрипторИсточникаСообщения*, *ИмяОбработчика*). Макрокоманда включается в карту сообщений того окна, которое содержит обработчик для его обслуживания. 2. Создать обработчик сообщения с прототипом afx_msg void *ИмяОбработчика*(). Большинство обработчиков имеют стандартные прототипы, описанные, например, в классе CWnd.

Например, нажатие кнопки ОК (с дескриптором IDOK) диалогового окна (класс MY_DIALOG) приводит к посылке сообщения типа WM_COMMAND. В качестве обработчика может использоваться функция вида afx_msg void *MY_DIALOG*::*OnButtonOK*( ) {…} . А макрокоманда включения выглядит как ON_COMMAND(*IDOK*, *OnButtonOK*).

Класс CDialog._Диалоговые окна функционально базируются на классе CDialog, производном от CWnd. По поведению бывают двух типов: модальные, которые перехватывают фокус и требуют завершения для дальнейшей работы приложения, и немодальные, позволяющие приложению работать без закрытия самого окна. Диалоговое окно получает сообщения от Windows, включая сообщения от элементов управления (ЭУ) диалогового окна. Таким образом, пользователи осуществляют интерактивное взаимодействие с приложением. Подключается #include <afxwin.h> .

По функциональному назначению диалоговые окна бывают пользовательские и стандартные, производные от класса CDialog и представленные в таблице 24 ниже.

Таблица 24 - Стандартные диалоги

Класс стандартного окна	Назначение
CColorDialog	выбор цветов
CFileDialog	выбор имени файла для открытия или сохранения
CFindReplaceDialog	управление поиском и заменой фрагментов в текстовом файле
CFontDialog	выбор фонта
CPrintDialog	настройка печати

Стиль конкретного диалогового окна определяется комбинацией стилей windows-окон и стилей диалоговых окон.

Члены класса (рисунок 39). Это конструкторы CDialog, методы инициализации немодального окна Create, CreateIndirect, модального окна InitModalIndirect. Методы DoModal, EndDialog для активизации и закрытия модального окна, void CDialog::GotoDlgCtrl (CWnd* УказательЭУ_ПолучателяФокуса) для передачи фокуса конкретному ЭУ окна и др. Кроме этого класс включает подменяемые методы: OnOk(), OnCancel(), OnInitDialog(). Диаграмма класса с учетом основных открытых членов представлена на рисунке ниже.

Создание и удаление диалоговых окон. Объект CDialog есть комбинация шаблона окна и CDialog-производного класса. Шаблон может быть выполнен: а) в виде ресурса - диалоговое окно визуально в редакторе ресурсов; б) в виде ресурса - диалоговое окно текстовым описанием; в) шаблон может быть создан программно в памяти.

В пользовательском классе, как правило, необходимо добавить члены-данные для поддержки ЭУ (ввода-вывода данных); обработчики сообщений ЭУ; предусмотреть инициализацию (обработка сообщения WM_INITDIALOG); описать конструкторы.

Ниже дается сводка наиболее часто используемых для этого методов.

1. Конструкторы CDialog-объектов

CDialog(LPCTSTR *ПоименованныйДескрипторОкна*, CWnd* *ОкноРодитель* = NULL);
CDialog(UINT *ДескрипторОкна*, CWnd* *ОкноРодитель* = NULL);
CDialog( );

Здесь *ОкноРодитель* указывает на родительское окно, к которому относится данное. Если NULL, то это главное окно приложения.

2. Метод virtual int DoModal (); вызывает модальное диалоговое окно. Метод void EndDialog(int *ПризнакЗавершения*); закрывает диалоговое окно и возвращает *ПризнакЗавершения* в точку вызова окна по DoModal ().

3. Метод инициализации окна, который требует подмены (overriding) и вызывается сообщением WM_INITDIALOG, которое посылается окну во время вызова Create, CreateIndirect или DoModal немедленно до вывода окна на экран virtual BOOL OnInitDialog( ); .

CDialog
CDialog (LPCTSTR lpszDialName, CWnd* Owner = NULL); CDialog (UINT nID, CWnd* Owner = NULL); CDialog ( ); virtual int DoModal ( ); void EndDialog (int Status); virtual BOOL OnInitDialog ( ); virtual void OnCancel ( ); virtual void OnOK ( ); void CDialog::GotoDlgCtrl (CWnd* УказательЭУ_ПолучателяФокуса); CWnd* CWnd::GetDlgItem (int ДескрипторЭУ) const; void CWnd::GetDlgItem (int ДескрипторЭУ, HWND* Указатель) const; void CWnd::SetWindowText (LPCTSTR СтрокаВывода) void CWnd::GetWindowText (CString& СтрокаВвода) const; int CWnd::GetDlgItemText(int ДескрипторЭУ, LPTSTR БуферПриемникСтроки, int ЧислоЧитаемыхСимволов) const; CWnd::DestroyWindow ( );
Класс MFC для создания производных классов пользовательских диалоговых объектов

Рисунок 39 - Диаграмма класса CDialog

При подмене первым вызывается метод CDialog::OnInitDialog(), а затем программируются действия по специфичной инициализации окна. Примерный текст представлен ниже:

```
BOOL МОЙ_ДИАЛОГ::OnInitDialog()
{
  CDialog::OnInitDialog();
  ……..
  // Действия по инициализации
  return TRUE;
}  .
```

4. Стандартный обработчик закрытия окна (с кодом возврата DoModal - IDOK) virtual void OnOK(); и обработчик закрытия окна (с кодом возврата DoModal – IDCANCEL) virtual void OnCancel(); .

Пользовательский класс может быть создан: а) с помощью мастера ClassWizard, который позволяет просматривать списки сообщений, генерируемых ЭУ окна, выбирать те, которые представляют интерес (при этом ClassWizard автоматически генерирует макрокоманды для карты сообщений окна и прототипы обработчиков, а программист дописывает реализации обработчиков), добавлять члены-данные для поддержки ЭУ; б) вручную.

Для конструирования модального окна на базе ресурса (шаблона) можно использовать любой public конструктор, а затем активизировать его DoModal(). Соответственно для создания модального окна надо: 1) создать пользовательский класс, производный от CDialog; 2) создать, используя конструктор, объект; 3) активизировать окно с ЭУ методом DoModal.

Для конструирования немодального окна надо использовать protected конструктор. Т.е. надо создать производный пользовательский класс диалогового окна, перегрузить конструктор. Далее создать объект, вызвав конструктор, создать на базе ресурса диалоговое окно методом Create. Соответственно для создания немодального окна надо: 1) создать пользовательский класс, производный от CDialog; 2) в его конструкторе вызвать Create; 3) создать окно, используя конструктор.

Если шаблон окна был создан в памяти, то используется структура данных типа DLGTEMPLATE. Она определяет координаты x, y, ширину и высоту окна cx, cy, стиль окна style (как комбинацию стилей окон и диалоговых окон), расширенный стиль dwExtendedStyle, число ЭУ в составе диалогового окна cdit и, следовательно, число структур типа DLGITEMTEMPLATE для их описания:

```
typedef struct
{
  DWORD style;
  DWORD dwExtendedStyle;
  WORD  cdit;
  short x;
  short y;
  short cx;
  short cy;
} DLGTEMPLATE, *LPDLGTEMPLATE; .
```

Здесь после создания CDialog-объекта используется метод CreateIndirect для создания немодального окна или InitModalIndirect и DoModal для создания модального окна.

Для немодальных окон следует также перегрузить обработчик OnCancel, вызвав из него DestroyWindow. Нельзя использовать CDialog::OnCancel без перегрузки, так там вызывается метод EndDialog, который делает окно невидимым, но не разрущает его.

Обрабатываемые сообщения. Это, в первую очередь, сообщения, обрабатываемые в объектах CWnd. Специфические сообщения объектов CDialog: а) сообщение WM_INITDIALOG, которое посылается от ОС окну во время вызова методов Create, CreateIndirect или DoModal немедленно до вывода окна на экран. Прототип обработчика

```
virtual BOOL CDialog::OnInitDialog();
```

б) сообщение от кнопок типа WM_COMMAND. Прототипы обработчиков

```
virtual void OnOK();
virtual void OnCancel();   .
```

2 ДО и ЭУ для ввода-вывода данных

Общая технология организации интерактивных графических интерфейсов на базе диалоговых окон и ЭУ предполагает набор следующих типовых действий.

А. РАЗРАБОТКА ОКОННОГО ИНТЕРФЕЙСА и в том числе:

а) проектирование оконного интерфейса, в т.ч. системы окон, состава их элементов (включая элементы управления – ЭУ), порядка их взаимодействия и использования;

б) для каждого диалогового окна проектирование элементов управления (например, кнопок, окон редактирования и т.д.), определение их вида, размещения, возлагаемых функций;

в) для каждого ЭУ добавление, например, в редакторе ресурсов, к шаблону (ресурсу) ОКНО_ПОДЛОЖКИ (основному или диалоговому окну), настройка свойств, получение идентификатора (дескриптора ID);

г) для каждого ЭУ внесение изменений в класс ОКНО_ПОДЛОЖКИ – например, описание прототипов функций-обработчиков сообщений;

д) внесение изменений в функции инициализации класса ОКНО_ПОДЛОЖКИ, например, при необходимости конкретной инициализации содержимого и свойств ЭУ в момент запуска экземпляра класса ОКНО_ПОДЛОЖКИ;

е) для каждого ЭУ описание функциональности - алгоритмов использования ЭУ для ввода-вывода информации в соответствующих функциях-обработчиках сообщений и т.д.

Б. ОРГАНИЗАЦИЯ ОБРАБОТКИ СООБЩЕНИЙ, посылаемых окну (здесь объекту класса ОКНО_ПОДЛОЖКИ - CDialog). Предполагает:

а) включение чувствительности окна (объекта соответствующего класса) к сообщениям с помощью макрокоманд в карте сообщений окна

ON_*СООБЩЕНИЕ* (*ID_ЭУ_ИсточникаСообщения*,
ИмяОбработчикаОкнаПриемникаСообщения) ;

б) описание обработчиков сообщений с прототипами

afx_msg void *ИмяОбработчикаОкнаПриемникаСообщения* (); .

В. РЕАЛИЗАЦИЯ ВВОДА-ВЫВОДА. Специфика выполнения – алгоритмы реализации ввода-вывода данных существенно зависит от особенностей функционирования ЭУ разных типов. Тем не менее, предполагает выполнение некоторых типовых действий.

1. ПОЛУЧЕНИЕ ДОСТУПА К ЭУ. Выполняется, например, путем получения указателя на ЭУ (например, окно редактирования класса CEdit или на дочернее окно диалогового окна или окна) с использованием метода класса CWnd ПОЛУЧИТЬ_ЭУ (ДескрипторЭУ, Указатель)

void CWnd::GetDlgItem (int *ДескрипторЭУ*, HWND* *Указатель*) const;

или метода класса CWnd УКАЗАТЕЛЬ (ДескрипторЭУ)

CWnd* CWnd::GetDlgItem (int *ДескрипторЭУ*) const; .

Особенность метода – тип указателя следует привести к типу соответствующего ЭУ; сам указатель временный и не может быть сохранен. При отрицательном результате указатель равен NULL.

ПРИМЕР использования для ЭУ – окно редактирования:

```
CEdit*   УказательЭУ;
УказательЭУ = ( CEdit* ) GetDlgItem ( ДескрипторЭУ );
GotoDlgCtrl ( УказательЭУ );  .
```

2. ПЕРЕДАЧА ФОКУСА ЭУ с использованием метода класса CDialog ПЕРЕЙТИ_К_ЭУ (*УказательЭУ_ПолучателяФокуса*)

void CDialog::GotoDlgCtrl (CWnd* *УказательЭУ_ПолучателяФокуса*); .

3. ПРЕОБРАЗОВАНИЕ ДАННЫХ В СТРОКОВЫЙ ТИП. Вывод данных, как правило, производится в строковом формате, что может потребовать их предварительного переформатирования. Так для преобразования в строку данных целого типа можно использовать функцию

wsprintf (*СтроковоеПредставлениеЧисла*, "%d", *ИсходноеЧисло*); .

Для преобразования вещественного значения можно использовать аналогичную функцию sprintf() либо функцию ПРЕОБРАЗОВАТЬ_В_СТРОКУ (*ИсходноеЧисло*, *ДлинаСтроки*, *СтроковоеПредставлениеЧисла*)

char* _gcvt (double, int, char) .

4. ВЫВОД ДАННЫХ:

а) ЭУ в значительной мере используют функциональность базового класса CWnd. Для вывода данных в окно (например, при работе с однострочным окном редактирования, а также для вывода полного содержимого многострочного окна) можно использовать метод УСТАНОВИТЬ_ТЕКСТ_В_ОКНЕ (*СтрокаВывода*)

void CWnd::SetWindowText (LPCTSTR *СтрокаВывода*) .

Метод устанавливает заголовок окна заданной строкой, а если окно – ЭУ, то устанавливает текст внутри окна. СтрокаВывода – Cstring-строка или С-строка. Метод вызывает посылку этому окну сообщения WM_SETTEXT.

ПРИМЕР вывода информации (числовых и не числовых данных) в виде строк в стандартном элементе управления – в окне редактирования (класс CEdit) приведен ниже. Пусть дескриптор окна редактирования - IDC_EDIT2. Тогда вывод данного в виде строки

```
CEdit  *pEditBox = ( CEdit * )  CDialog:: GetDlgItem ( IDC_EDIT2 );
pEditBox -> SetWindowText ( OutputStr );  .
```

Вывод данного целого типа

```
wsprintf ( OutputStr, "%d", IntNumber );
CEdit  *pEditBox = ( CEdit * ) CDialog:: GetDlgItem ( IDC_EDIT2 );
pEditBox -> SetWindowText( OutputStr );
```

Вывод данного вещественного типа (фрагмент текста программы)

```
#include <stdio.h>
……
float FloatNumber = 12.345;
char OutputStr [100];
CEdit  *pEditBox = ( CEdit * )  CDialog:: GetDlgItem ( IDC_EDIT2 );
pEditBox -> SetWindowText ( OutputStr );  .
……..
sprintf ( OutputStr, "%f" , FloatNumber );
pEditBox -> SetWindowText ( OutputStr );  .
……
_gcvt (FloatNumber, strlen(OutputStr), OutputStr );
pEditBox -> SetWindowText ( OutputStr );
……
```

б) кроме этого для вывода могут использоваться методы

```
CWnd::SetDlgItemText (…),
CWnd::SetDlgItemInt (…)  .
```

в) для работы с ЭУ конкретных типов можно использовать специфические методы. Например, для вывода в многострочное окно редактирования следует применять методы CEdit типа: CEdit::SetLine(…) , CEdit::ReplaceSel(…) .

5. ПРЕОБРАЗОВАНИЕ ДАННЫХ СТРОКОВОГО ТИПА. Ввод данных, как правило, производится в строковом формате, что может потребовать их последующего переформатирования. Так для получения из строки данного вещественного типа можно использовать функцию

Число = atof (*СтроковоеПредставлениеЧисла*).

6. ВВОД ДАННЫХ.

а) для ввода данных из окна (например, при работе с однострочным окном редактирования, а также для считывания полного содержимого многострочного окна) можно использовать метод CWnd ПОЛУЧИТЬ_ТЕКСТ_ИЗ_ОКНА (*СтрокаВвода*)

void CWnd::GetWindowText (CString& *СтрокаВвода*) const;

или метод ПОЛУЧИТЬ_ТЕКСТ_ИЗ_ОКНА (*СтрокаВвода*)

int CWnd::GetWindowText (LPCTSTR *СтрокаВвода*, int *ЧислоЧитаемыхСимволов*) const; .

Методы копируют заголовок окна (класс CWnd) в заданную строку, а если окно – ЭУ, то копируется содержимое окна. *СтрокаВывода* здесь Cstring-строка или С-строка. Метод вызывает посылку этому окну сообщения WM_GETTEXT.

ПРИМЕР (фрагмент) ввода данных строкового, целого и вещественного типов в окне редактирования с идентификатором IDC_EDIT1 представлен ниже.

```
char szMyString [80] = " ";
int IntNumber ;
float FloatNumber ;
……..
CEdit  *pEditBox = (CEdit *) CDialog:: GetDlgItem ( IDC_EDIT1 );
pEditBox -> GetWindowText  ( szMyString, sizeof  szMyString - 1);
……..
CEdit  *pEditBox = ( CEdit * ) CDialog:: GetDlgItem ( IDC_EDIT1 );
pEditBox -> GetWindowText  ( szMyString, sizeof szMyString - 1);
IntNumber = atof (szMyString);
……..
CEdit  *pEditBox = ( CEdit * ) CDialog:: GetDlgItem ( IDC_EDIT1 );
pEditBox -> GetWindowText  ( szMyString, sizeof  szMyString - 1);
FloatNumber = atof (szMyString);
……..
```

б) ниже представлены другие методы CWnd для получения текста названия (заголовка title) или текста, ассоциируемого с ЭУ в диалоговом окне. Они копируют текст в заданную строку ПОЛУЧИТЬ_ТЕКСТ_ИЗ_ЭУ (*ДескрипторЭУ*, *БуферПриемникСтроки*, *ЧислоЧитаемыхСимволов*)

int CWnd::GetDlgItemText(int *ДескрипторЭУ*, LPTSTR *БуферПриемникСтроки*, int *ЧислоЧитаемыхСимволов*) const;

или ПОЛУЧИТЬ_ТЕКСТ_ИЗ_ЭУ (*ДескрипторЭУ*, *БуферПриемникСтроки*)

int CWnd::GetDlgItemText(int *ДескрипторЭУ*, CString& *БуферПриемникСтроки*) const;
.

ПРИМЕР использования

```
char MyString[20];
```

```
float MyFloat;
……..
GetDlgItemText ( IDC_EDIT1, MyString,15);
MyFloat = atof ( MyString );  .
```

в) для ввода данных целого типа может использоваться метод ВВЕСТИ_ЦЕЛОЕ_ЧИСЛО_ИЗ_ЭУ (*ДескрипторЭУ*, = NULL, *НаличиеЗнака*)

UINT CWnd::GetDlgItemInt (int *ДескрипторЭУ*, BOOL* *lpTrans* = NULL, BOOL *НаличиеЗнака* = TRUE) const;

Если признак *НаличиеЗнака* = 1, то результат ввода преобразуется в целое со знаком. А если 0, то результат ввода преобразуется в целое без знака.

г) для работы с ЭУ конкретных типов можно использовать специфические методы. Например, для ввода из многострочного окна редактирования следует применять методы:

int CEdit::GetLine(int *НомерСтрокиОкнаРедактирования*, LPTSTR *СтрокаПри-емник*) const;
int CEdit::GetLine(int *НомерСтрокиОкнаРедактирования*, LPTSTR *СтрокаПри-емник*, int *МаксимальнаяДлинаСтрокиПриемника*) const; .

ПРИМЕР использования:

```
CEdit*      МоеОкно;
int         НомерСтроки;
int         ЧислоСтрок = МоеОкно -> GetLineCount ( );
CString     strText, strLine;

for (НомерСтроки = 0; НомерСтроки <  ЧислоСтрок;  НомерСтроки ++)
{
    МоеОкно -> GetLine(НомерСтроки, strText.GetBuffer ( МоеОкно -> LineLength  ( НомерСтроки ) ) );
   strText.ReleaseBuffer ( );
    …
}  .
```

Класс CButton. Общее описание, назначение. Элемент управления типа “кнопка” (button) – небольшое прямоугольное (квадратное) производное окно. Кнопку, путем щелчка, можно нажать или отжать. При этом она, как правило, меняется внешне. Может использоваться автономно либо в составе группы кнопок. Подключается как #include <afxwin.h>. Функциональность кнопок поддерживается классом CButton, производным от CWnd.

Стили. Стиль кнопки зависит от параметров стиля, задаваемых, например, при инициализации соответствующего объекта класса CButton методом Create. Либо задается установкой флажков в окнах свойств ресурса в редакторе ресурсов. Разновидности кнопок (класс CButton): переключатели (check box - стиль BS_CHECKBOX), радио кнопки (radio button - стиль BS_RADIOBUTTON) командные кнопки (pushbutton - стиль BS_PUSHBUTTON). Еще один вид – графическая кнопка (класс CBitmapButton произ-

водный от CButton). Это кнопка, маркируемая изображением (типа bitmap) вместо текста. Может маркироваться разными изображениями для разных состояний кнопки: нажата, отжата, выбрана (focused), недоступна (disabled) и т.д. Ниже представлены основные стили (таблица 25):

- BS_PUSHBUTTON - командная кнопка, которая посылает сообщения типа WM_COMMAND своему окну-владельцу (owner window) каждый раз, когда пользователь выбирает кнопку;
- BS_CHECKBOX - переключатель. Это кнопка в виде небольшого прямоугольника с поясняющим текстом справа (по умолчанию) либо слева;
- BS_RADIOBUTTON - радио-кнопка. Это кнопка в виде небольшой окружности с поясняющим текстом справа (по умолчанию) либо слева. Используются, как правило, группами связанных и взаимоисключающих по выбору кнопок;
- BS_LEFTTEXT - в комбинации с переключателями или радио-кнопками позволяет выводить поясняющий текст слева от кнопки;
- BS_AUTOCHECKBOX - разновидность переключателя с отметкой, которая появляется в его поле при выборе пользователем;
- BS_AUTORADIOBUTTON - разновидность радио-кнопки, которая выделяется пользователем и автоматически снимает выделение со всех других кнопок этого стиля, объединенных в группу;
- BS_GROUPBOX - прямоугольник, в котором группируются другие кнопки;
- BS_DEFPUSHBUTTON - кнопка с темной рамкой по периметру. Пользователь может выбирать ее нажатием клавиши ENTER.

Члены класса. Конструктор CButton, метод инициализации Create, методы GetState, SetState для получения информации о текущем состоянии кнопки либо для программной установки состояния и др.

Создание и удаление. Кнопки могут создаваться на основе соответствующего оконного шаблона (dialog template) либо непосредственно программным путем. В первом случае кнопка создается визуально в редакторе ресурсов либо описывается в текстовом редакторе. В обоих случаях далее создается объект класса CButton соответствующим конструктором CButton *МояКнопка*. При этом при ресурсном описании кнопки ее параметры автоматически инициализируют создаваемый объект. При программном создании для этого используется метод Create. Он создает кнопку и “прикрепляет” ее к CButton-объекту. Прототип метода

BOOL CButton::Create (LPCTSTR *ИмяКнопки*, DWORD *СтильКнопки*, const RECT& *РазмерКоординаты*, CWnd* *РодительскоеОкно*, UINT *ДескрипторКнопки*);

Здесь возвращаемое значение – признак завершения (нормальное завершение – ненулевой код); *РодительскоеОкно,* как правило, класса CDialog. Используемые в методе стили: WS_CHILD (всегда), WS_VISIBLE (как правило), WS_DISABLED (иногда), WS_GROUP (для групп кнопок) и др. Стиль WS_VISIBLE определяет, что ОС Windows посылает кнопке все сообщения, требуемые для ее активизации, визуализации.

Пример использования метода

CButton *МояКомандная Кнопка*, *МояРадиоКнопка*;

МояКомандная Кнопка.Create ("Командная Кнопка"), WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON, CRect(5,5,100,30), *Указатель*, ID_PushButton);

МояРадиоКнопка.Create (("РадиоКнопка"), WS_CHILD | WS_VISIBLE | BS_RADIOBUTTON, CRect(10,40,100,70), *Указатель*, ID_RadioButton); .

Кнопка (объект класса CButton), созданная первым способом (на основе диалогового, оконного ресурса), разрушается автоматически, когда пользователь закрывает диалоговое окно. Если кнопка создавалась программно и к тому же в динамической памяти (heap) с помощью оператора new, то необходимо удалить объект оператором delete. Если же объект создан в стеке (stack) или был внедрен в родительский диалоговый объект, то он разрушается автоматически.

Можно создать кнопка независимо от диалогового или любого другого окна, например,

```
ДескрипторКнопки = CreateWindow(
   "BUTTON",   "OK",    WS_VISIBLE | WS_CHILD | BS_DEFPUSHBUTTON,
   10,   10,   100,   100,
   ДескрипторОРодительскогоОкна,  NULL,
   (HINSTANCE) GetWindowLong (hwnd, GWL_HINSTANCE),   NULL);     .
```

Посылаемые сообщения. Список сообщений: BM_CLICK, BM_GETCHECK, BM_GETIMAGE, BM_GETSTATE, BM_SETCHECK, BM_SETIMAGE, BM_SETSTATE, BM_SETSTYLE, BN_CLICKED, BN_DBLCLK, BN_DOUBLECLICKED (аналог BN_DBLCLK), BN_KILLFOCUS, BN_SETFOCUS, WM_CTLCOLORBTN и др. Список сообщений можно получить в MSDN (см. Buttons Overview, Buttons Messages и т.д.), а информацию о сообщении путем поиска, например, по образцу BN_CLICKED.

Сообщение BN_CLICKED посылается, когда пользователь “щелкает” кнопку. Родительское окно кнопки получает его (код сообщения) через сообщение WM_COMMAND. При этом в обработчик окна поступает следующая информация:

```
LRESULT CALLBACK WindowProc (
  HWND ДескрипторДиалоговогоОкнаРодителя,
  UINT КодПолученногоСообщения,
  WPARAM АтрибутыПолученногоСообщения,
  LPARAM ДескрипторКнопкиИсточникаСообщения
)   .
```

Здесь *КодПолученногоСообщения* - WM_COMMAND; младшее слово параметра *АтрибутыПолученногоСообщения* содержит идентификатор кнопки, а старшее слово специфицирует сообщение как BN_CLICKED. Недоступная кнопка (стиль disabled) такое сообщение окну-родителю не посылает.

Сообщение BN_DOUBLECLICKED автоматически посылается, когда пользователь дважды “щелкает” кнопку, если включен стиль BS_USERBUTTON, BS_RADIOBUTTON, BS_OWNERDRAW. Другие кнопки посылают это сообщение только при включенном стиле BS_NOTIFY. При этом родительское окно кнопки получает его (код сообщения) через сообщение WM_COMMAND: здесь *КодПолученногоСообщения* - WM_COMMAND; младшее слово параметра *АтрибутыПолученногоСообщения* содержит идентификатор кнопки, а старшее слово специфицирует сообщение как BN_CLICKED. Недоступная кнопка (стиль disabled) такое сообщение окну-родителю не посылает.

Обработка сообщений. Необходимо связать ЭУ-источник сообщения и обработчик сообщения: описать обработчик сообщения; включить чувствительность родителя к сообщению – задать макрокоманду карты сообщений. Например,

ON_BN_CLICKED (ID_ЭУ_ИсточникаСообщения, ИмяОбработчикаОкнаПриемникаСообщения).

Стили командных кнопок приведены ниже в таблице 25.

Таблица 25 - Стили командных кнопок

Параметр стиля	Описание
BS_BOTTOM	размещает текст вверху прямоугольника кнопки
BS_CENTER	центрирует текст по горизонтали в прямоугольнике кнопки
BS_DEFPUSHBUTTON	кнопка с жирной черной рамкой. Если она в составе диалогового окна, то пользователь может выбирать ее нажатием клавиши ENTER, даже если она не в фокусе ввода. Полезно для быстрого выбора наиболее используемой кнопки (или кнопки по умолчанию)
BS_LEFT	выравнивает текст влево в прямоугольнике кнопки
BS_PUSHBUTTON	командная кнопка, посылающая при “щелчке” пользователя сообщение WM_COMMAND окну-собственнику, где размещена эта кнопка
BS_RIGHT	выравнивает текст вправо в прямоугольнике кнопки
BS_TOP	размещает текст вверху прямоугольника кнопки
BS_VCENTER	центрирует текст по вертикали в прямоугольнике кнопки
BS_RADIOBUTTON	круглая кнопка, посылающая при “щелчке” пользователя сообщение WM_COMMAND окну-собственнику, где размещена эта кнопка. При повторном “щелчке” выбор отменяется
BS_CHECKBOX	прямоугольная кнопка, которую можно выбрать (переключатель)
BS_GROUPBOX	прямоугольная область, охватывающая группу кнопок

Класс CEdit. Назначение, общее описание. Элемент управления типа “окно редактирования” (edit) – небольшое прямоугольное (квадратное) дочернее (child) окно, в котором пользователь может вводить текст с клавиатуры. Фокус передается окну “мышью”, клавишей Tab. Работает как редактор текста, позволяя вводить, корректировать, копировать, вставлять, вырезать текст (взаимодействуя с clipboard). Подключается #include <afxwin.h>.

Функциональность кнопок поддерживается классом CEdit, производным от CWnd.

Стили. Основные разновидности окна редактирования (класс CEdit): однострочный редактор (по умолчанию); многострочный редактор (стиль ES_MULTILINE); окно только для отображения текста (стиль ES_READONLY); окно для ввода скрытой информации (стиль ES_PASSWORD). Режим использования окна зависит от параметров стиля, задаваемых, например, при инициализации соответствующего объекта класса CEdit методом Create либо задается установкой флажков в окнах свойств ресурса в редакторе ресурсов. В многострочном режиме работа окна зависит также от включения-выключения возможности горизонтального и вертикального скроллирования текста. Стили окон редактирования приведены ниже в таблице 26.

Члены класса. Это конструктор CEdit, метод инициализации Create. Методы для получения значений атрибутов окна редактирования в одно и многострочном вариантах использования. Например, GetSel для получения начальной и конечной позиций текущего выделения в окне

CEdit* *МоеОкно*;
...
DWORD *ПозицииВыделения* = *МоеОкно* -> GetSel (); .

Методы для установки значений атрибутов окна редактирования, работы с окном в одно и многострочном вариантах использования. Например, SetSel для выделения фрагмента текста по заданным начальной и конечной позициям выделения, например

МоеОкно -> SetSel (*НачальнаяПозиция*, *КонечнаяПозиция*); .

Методы Clear, Cut, Undo и др., поддерживающие работу с clipboard.

Методы для чтения-записи строк. Объекты класса CEdit в значительной мере используют функциональность базового класса CWnd. Так при работе с однострочным окном следует использовать методы (например, для установки текста в окне редактирования, а также для считывания ранее введенного текста):

CWnd::GetWindowText (…) ,
CWnd::SetWindowText (…) .

Они же могут применяться для вывода-чтения полного содержимого многострочного окна. Для работы с частями текста многострочного окна можно применять методы CEdit:

int CEdit::GetLine(int *НомерСтрокиОкнаРедактирования*, LPTSTR *СтрокаПри-емник*) const;
int CEdit::GetLine(int *НомерСтрокиОкнаРедактирования*, LPTSTR *СтрокаПри-емник*, int *МаксимальнаяДлинаСтрокиПриемника*) const;
CEdit::ReplaceSel(…) .

Создание и удаление. Окна редактирования могут создаваться на основе соответствующего оконного шаблона (dialog template) либо непосредственно программным путем. В первом случае окно создается визуально в редакторе ресурсов либо описывается в текстовом редакторе. В обоих случаях далее создается объект класса CEdit соответствующим конструктором CEdit - CEdit *МоеОкноРедактирования*; .

При этом при ресурсном описании окна его параметры автоматически инициализируют создаваемый объект. При программном создании используется метод Create. Он создает окно и “прикрепляет” его к CEdit-объекту. Для этого следует вставить вызов Create в соответствующий конструктор. Прототип Create

BOOL CEdit::Create (DWORD *СтильОкна*, const RECT& *РазмерКоординаты*, CWnd* *РодительскоеОкно*, UINT *ДескрипторОкна*); .

Здесь: возвращаемое значение – признак завершения (нормальному завершению соответствует ненулевой код); *РодительскоеОкно*, как правило, класса CDialog. Используемые в методе стили: WS_CHILD (всегда), WS_VISIBLE (как правило), WS_DISABLED (иногда), WS_GROUP (для групп ЭУ) и др. Стиль WS_VISIBLE определяет, что ОС Windows посылает окну все сообщения, требуемые для его активизации, визуализации.

Пример использования метода

CEdit *МоеОкно*;

или

CEdit* *УказательМоеОкно* = new CEdit;

УказательМоеОкно -> Create (ES_MULTILINE | WS_CHILD | WS_VISIBLE | WS_TABSTOP | WS_BORDER, CRect (5, 5, 100, 300), this, 1); .

Окно редактирования (объект класса CEdit), созданное в составе диалогового, оконного ресурса, разрушается автоматически, когда пользователь закрывает диалоговое окно. Если окно создавалась программно и к тому же в динамической памяти (heap) с помощью оператора new, то необходимо удалить объект оператором delete. Если же объект создан в стеке (stack) или был внедрен в родительский диалоговый объект, то он разрушается автоматически.

Посылаемые сообщения. Список сообщений можно получить в MSDN, а информацию о сообщении путем поиска, например, по образцу EM_GETLINE. Ниже дана общая характеристика групп сообщений.

1. Сообщения (notification message), генерируемые самим окном редактирования и посылаемые ОС через сообщение WM_COMMAND: типа EN_CHANGE, EN_KILLFOCUS, EN_SETFOCUS, EN_UPDATE, EN_VSCROLL и др. Например, сообщение EN_CHANGE генерируется и посылается окном, если пользователь выполнил действия, которые могут привести к событию “изменение текста в окне”. В отличие от EN_UPDATE, это сообщение посылается после того как система обновляет экран. При этом в обработчик окна поступает следующая информация

```
LRESULT CALLBACK WindowProc(
  HWND ДескрипторДиалоговогоОкнаРодителя,
  UINT КодПолученногоСообщения,
  WPARAM АтрибутыПолученногоСообщения,
  LPARAM ДескрипторОкошкаИсточникаСообщения
);  .
```

Здесь *КодПолученногоСообщения* - WM_COMMAND; младшее слово параметра *АтрибутыПолученногоСообщения* содержит идентификатор окна, а старшее слово специфицирует сообщение как EN_CHANGE.

Сообщение EN_UPDATE посылается, когда окно нуждается в перерисовке. Например, посылается после форматирования текста, но до вывода его в окне. Это делает возможным изменить размеры окна при необходимости.

2. Сообщения, адресуемые окну редактирования от программ приложения. Они посылаются принудительно, например, с помощью SendMessage и позволяют программно влиять на состояние окна редактирования. Это сообщения типа EM_GETLINE, EM_SETSEL и др. Например, сообщение EM_GETLINE посылается для вызова копирования строки текста из окна в буфер. Код функции представлен ниже

```
SendMessage(
  (HWND) ДескрипторОкнаНазначения,
  EM_GETLINE,
  (WPARAM) НомерЧитаемойСтроки,
  (LPARAM) УказательПриемникаСтроки          // line buffer (LPCTSTR)
);
```

В однострочном окне *НомерЧитаемойСтроки* игнорируется.

Сообщение EM_SETSEL посылается для выделения диапазона символов. Вид функции представлен ниже

SendMessage(
(HWND) *ДескрипторОкнаНазначения*,
EM_SETSEL,
(WPARAM) *НачальнаяПозицияВыделения*,
(LPARAM) *КонечнаяПозицияВыделения*
); .

3. После инициализации окна редактирования с использованием метода Create ОС посылает окну редактирования сообщения типа WM_CREATE, WM_NCCREATE и др. Они воспринимаются обработчиками-методами класса CWnd типа OnCreate(), OnNcCreate() и др. При необходимости методы можно подменить в пользовательском классе окна редактирования, производном от CEdit. Например, OnCreate() можно подменить для начальной инициализации пользовательского класса.

Примеры макрокоманд включения:

ON_EN_CHANGE(*ID_ЭУ_ИсточникаСообщения*,
ИмяОбработчикаОкнаПриемникаСообщения),
ON_EN_ UPDATE(*ID_ЭУ_ИсточникаСообщения*,
ИмяОбработчикаОкнаПриемникаСообщения).

Таблица 26 - Стили окон редактирования

Параметр стиля	Описание
ES_CENTER	центрирует текст
ES _LEFT	выравнивает текст по левому краю
ES _RIGHT	выравнивает текст по правому краю
ES _LOWERCASE	при вводе символы преобразуются в нижний регистр
ES _UPPERCASE	при вводе символы преобразуются в верхний регистр
ES _MULTILINE	обеспечивает работу в режиме многострочного редактора. При комбинации с другими стилями возможна вертикальная и горизонтальная прокрутка текста
ES _PASSWORD	вводимый текст скрывается символами “*”
ES _READONLY	текст отображается без возможности ввода или редактирования

3 Диалоговое окно в составе главного окна

ПРИМЕР. Создать MFC-приложение на базе ТКП с оконным интерфейсом из главного окна и диалогового окна с двумя кнопками OK, CANCEL. При запуске приложения должно выводиться главное окно и на фоне главного окна приложения - диалоговое окно, которому и передается управление. Диалоговое окно функционирует в модальном режиме и после нажатия любой из кнопок OK или CANCEL исчезает, возвращая управление главному окну. Диалоговое окно появляется каждый раз при событии “перерисовки” главного окна, т.е. при попытке изменения размеров главного окна.

Из описания задания следует:

1) начальная визуализация диалогового окна должна происходить автоматически при запуске приложения, как реакция приложения на сообщение WM_PAINT, посылаемое системой главному окну приложения при его запуске;

2) визуализация диалогового окна должна происходить автоматически каждый раз, как реакция приложения на сообщение WM_PAINT, посылаемое системой главному окну приложения при событии его “перерисовки”;

3) для обработки сообщений, связанных с нажатием указанных кнопок диалогового окна можно использовать стандартные обработчики сообщений.

Приложение создается на базе типового каркаса MFC-приложения. Для создания диалогового окна используется встроенный редактор ресурсов как альтернатива описанию ресурса - диалоговое окно текстом в файле ресурсов. ПОРЯДОК (схема) выполнения задачи представлен ниже.

1. Спроектировать интерфейс приложения (здесь диалоговое окно на фоне главного).

Определить состав элементов управления окна и их вид. При необходимости описать окна и ЭУ на соответствующем языке описания ресурсов. Здесь интерфейсные формы включают, как показано ниже (рисунок 40), главное окно и диалоговое окно, выводимое на фоне главного, с двумя командными кнопками ОК и CANCEL.

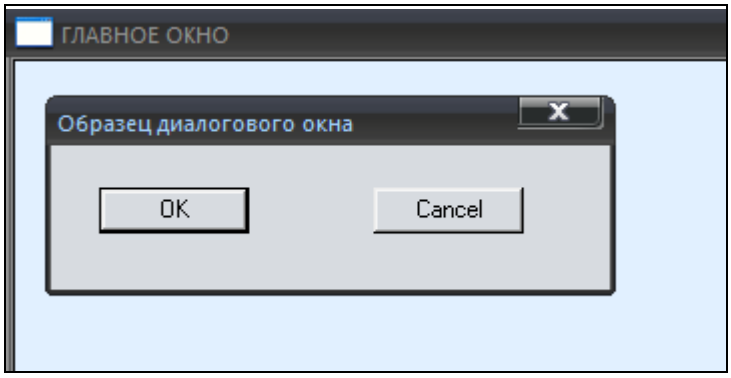


Рисунок 40 - Вид интерфейса

Примерный формат описания окна на языке описания ресурсов представлен ниже

ДескрипторОкна DIALOG DISCARDABLE *КоординатыОкна*
STYLE *СтильОкна1* | *СтильОкна2* | …
CAPTION *ЗаголовокОкна*
FONT *ИспользуемыйШрифт*
BEGIN
ОписаниеЭлементаУправления1
ОписаниеЭлементаУправления2
……..
END .

2. Специфицировать состав событий-сообщений при работе с диалоговым окном и реакций приложения на эти сообщения:

- событие “нажатие кнопки ОК” вызывает сообщение WM_COMMAND. Реакция стандартная – закрытие окна. Макрокоманда включения чувствительности окна к сообщениям этого типа

ON_COMMAND (IDOK, <*ИмяПользовательскогоОбработчика*>),
например ON_COMMAND (IDOK, OnButtonOK). Поскольку здесь используется стандартный обработчик, то не обязательно указывать явно этот тип сообщений в очереди обрабатываемых сообщений. Здесь IDOK – стандартный дескриптор кнопки ОК;

- событие “нажатие кнопки CANCEL”, вызывает сообщение WM_COMMAND. Реакция стандартная – закрытие окна. Макрокоманда включения чувствительности окна к сообщениям этого типа

ON_COMMAND (IDCANCEL, <*ИмяПользовательскогоОбработчика*>).

Например, ON_COMMAND (IDCANCEL, OnButtonCANCEL). Поскольку используется стандартный обработчик, то не обязательно указывать явно этот тип сообщений в очереди обрабатываемых сообщений. Здесь IDCANCEL – стандартный дескриптор кнопки CANCEL.

3. Специфицировать состав событий-сообщений при работе с главным окном и реакций приложения на эти сообщения: - события “запуск приложения с перерисовкой окна”, “изменение размеров окна”, “перекрытие окна” и др., приводят к посылке сообщений типа WM_PAINT. Реакция пользовательская – открытие (визуализация) диалогового окна с передачей ему управления. Макрокоманда включения чувствительности окна к сообщениям этого типа ON_WM_PAINT ( ). Поскольку здесь используется пользовательский обработчик afx_msg void OnPaint() { … }; , то в очереди обрабатываемых сообщений главного окна необходимо явно указать этот тип сообщений.

4. Спроектировать классы приложения. Здесь используются три пользовательских класса. Это класс class APPLICATION: public CWinApp для создания экземпляра приложения, класс class WINDOW: public CFrameWnd, для создания объекта – главное окно приложения, класс class MY_DIALOG: public CDialog для создания объекта – ДО. Состав и диаграммы классов, отображающие порядок наследования классов, используемых при создании приложения, представлены ниже (рисунки 41-43).

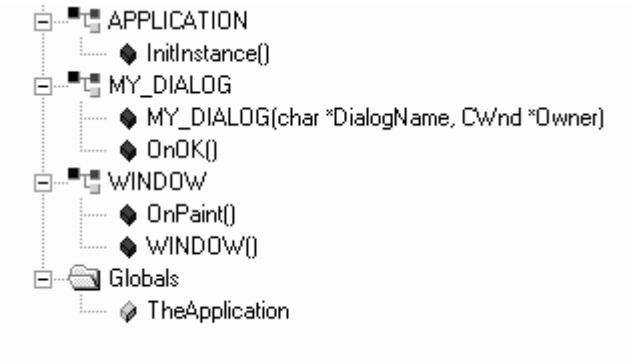


Рисунок 41 - Состав классов

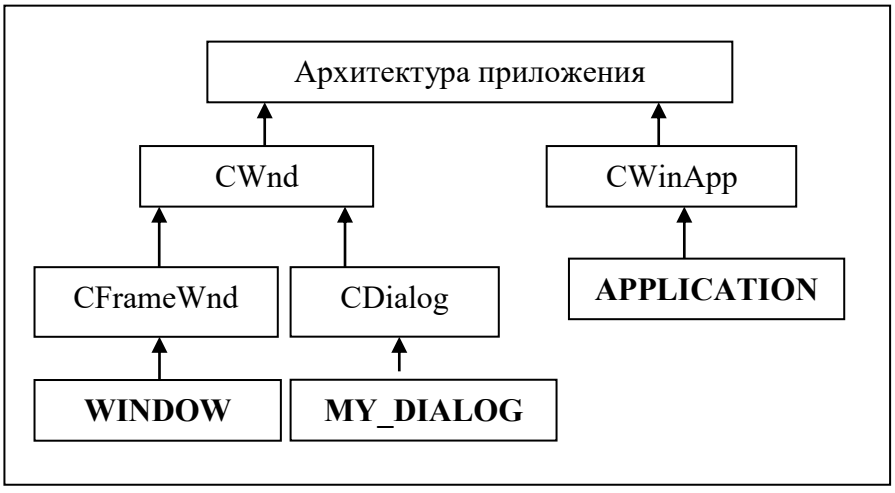


Рисунок 42 - Диаграмма классов

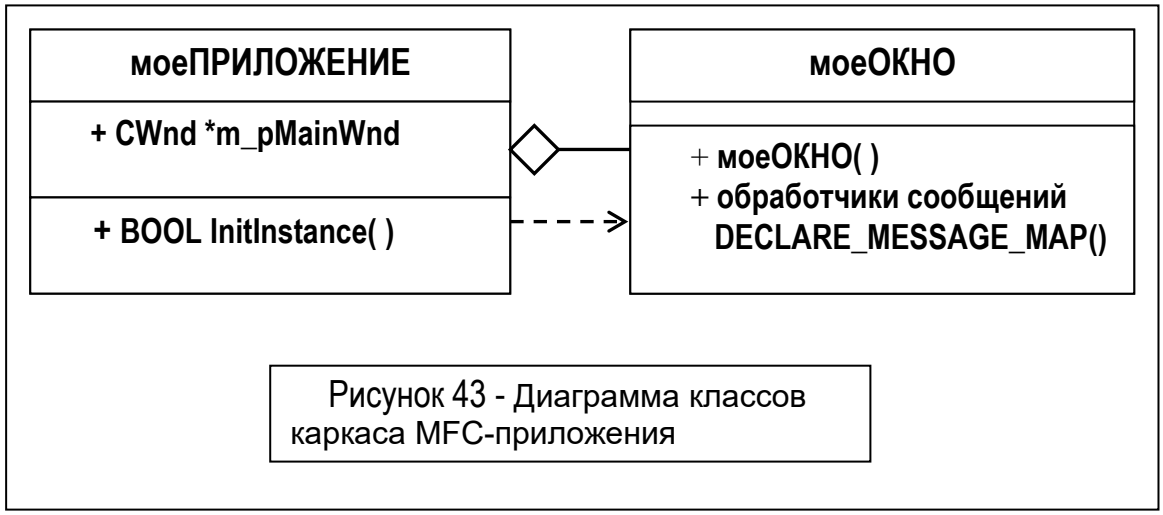


Рисунок 43 - Диаграмма классов каркаса MFC-приложения

5. Спроектировать модульную структуру приложения. Здесь для представления программной части (за исключением ресурсов) может использоваться один модуль.

6. Разработать соответствующий код. Ниже представлен исходный и модифицированный код приложения (с диалоговыми окнами, выводимыми до и в составе главного окна)

```
#include <afxwin.h>
#include <iostream>
#include "resource.h"
using namespace std;

//////////////////////////////////////////////////////
class MY_DIALOG : public CDialog
{
public:
   MY_DIALOG( char *DialogName, CWnd *Owner ):
              CDialog(DialogName, Owner)  {  };
   DECLARE_MESSAGE_MAP()
};

BEGIN_MESSAGE_MAP(MY_DIALOG, CDialog)
END_MESSAGE_MAP()

//////////////////////////////////////////////////////
class WINDOW : public CFrameWnd
{
public:
     WINDOW();
     afx_msg void OnPaint ( );
     DECLARE_MESSAGE_MAP()
};
```

```
    WINDOW::WINDOW ( )
    {
        Create( NULL,"ГЛАВНОЕ ОКНО" );
    }

    afx_msg void WINDOW::OnPaint  ()
    {
        CPaintDC dc(this);
        MY_DIALOG  TheDialog ( ( LPTSTR ) IDD_DIALOG1, this);
        TheDialog.DoModal ( );
    };

    BEGIN_MESSAGE_MAP(WINDOW, CFrameWnd)
       ON_WM_PAINT()
    END_MESSAGE_MAP()

    ////////////////////////////////////////////////
    class APPLICATION: public CWinApp
    {
    public:
        BOOL InitInstance ( );
    };

    BOOL APPLICATION::InitInstance ( )
    {
        m_pMainWnd =  new WINDOW;
        m_pMainWnd -> ShowWindow ( m_nCmdShow );
        m_pMainWnd -> UpdateWindow  ();
        return TRUE;
    }

    ////////////////////////////////////////////////////////
APPLICATION  TheApplication;  .

#include <afxwin.h>
#include <iostream>
#include "resource.h"
using namespace std;

////////////////////////////////////////////////
class MY_DIALOG : public CDialog
{
public:
   MY_DIALOG(char *DialogName, CWnd *Owner): CDialog(DialogName, Owner) { };
```

```
   DECLARE_MESSAGE_MAP()
};

BEGIN_MESSAGE_MAP(MY_DIALOG, CDialog)
END_MESSAGE_MAP()

//////////////////////////////////////////////////
class WINDOW : public CFrameWnd
{
public:
        WINDOW();
        afx_msg void OnPaint();
        DECLARE_MESSAGE_MAP()
};

WINDOW::WINDOW()
{
        Create(NULL,"ГЛАВНОЕ ОКНО ");
        //или MY_DIALOG TheDialog((LPTSTR)IDD_DIALOG1 ,this);
        //или TheDialog.DoModal();
}

afx_msg void WINDOW::OnPaint()
{
        CPaintDC dc(this);
        MY_DIALOG TheDialog((LPTSTR)IDD_DIALOG1,this);
        TheDialog.DoModal();
};

BEGIN_MESSAGE_MAP(WINDOW, CFrameWnd)
  ON_WM_PAINT()
END_MESSAGE_MAP()

//////////////////////////////////////////////////
class APPLICATION:public CWinApp
{
public:
        BOOL InitInstance();
};

BOOL APPLICATION::InitInstance()
{
        m_pMainWnd =  new WINDOW;
        //или без MY_DIALOG TheDialog((LPTSTR)IDD_DIALOG1 ,NULL);
        //или без TheDialog.DoModal();
```

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

//////////////////////////////////////////////////////////////
APPLICATION  TheApplication; .
```

ДО в составе главного окна. Переопределение обработчиков. ПРИМЕР. *Создать MFC-приложение на базе ТКП с оконным интерфейсом из главного окна и диалогового окна с двумя кнопками OK, CANCEL. При запуске приложения должно выводиться главное окно. При щелчке левой клавишей мыши в главном окне должно активизироваться диалоговое окно и выводиться на фоне главного окна приложения. Диалоговое окно функционирует в модальном режиме и после нажатия любой из кнопок (OK или CANCEL) исчезает, возвращая управление главному окну. При нажатии кнопки OK должно также выводиться сообщение, подтверждающее факт нажатия именно этой кнопки.* Из описания задания следует:

1) визуализация диалогового окна должна происходить каждый раз как реакция на сообщение главному окну типа WM_LBUTTONDOWN;

2) для обработки сообщений, связанных с нажатием кнопки CANCEL диалогового окна можно использовать стандартный обработчик сообщений;

3) для обработки сообщений, связанных с нажатием кнопки OK, используется пользовательский обработчик сообщений, подтверждающий с помощью функции MessageBox факт нажатия именно этой кнопки и завершающий работу диалогового окна.

Приложение создается на базе типового каркаса MFC-приложения. Для создания диалогового окна используется встроенный редактор ресурсов как альтернатива описанию ресурса - диалоговое окно текстом в файле ресурсов. ПОРЯДОК проектирования приложения представлен ниже.

1. Спроектировать интерфейс. Определить состав элементов управления окна и их вид. При необходимости описать окна и ЭУ на соответствующем языке описания ресурсов. Здесь интерфейсные формы включают, как показано ниже (рисунок 44), главное окно и диалоговое окно, выводимое на фоне главного, с двумя командными кнопками OK и CANCEL.

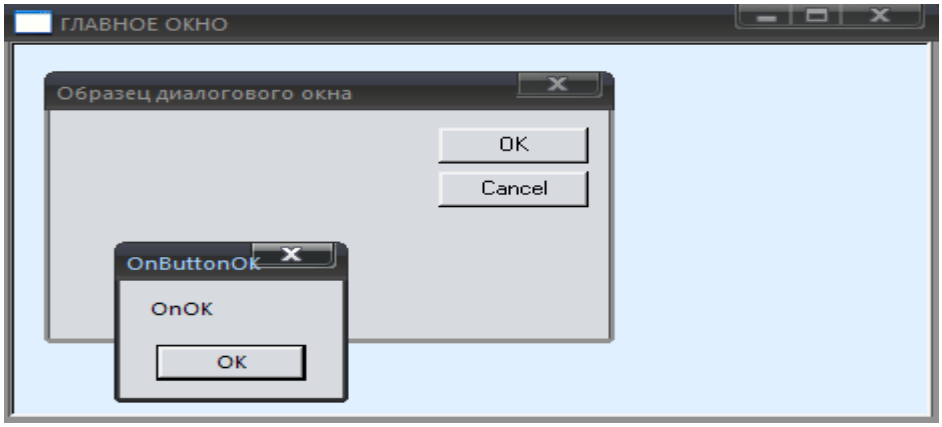


Рисунок 44 - Вид интерфейса

2. Специфицировать состав событий-сообщений при работе с диалоговым окном и реакций приложения на эти сообщения:

- событие “нажатие кнопки ОК” вызывает сообщение WM_COMMAND. Реакция пользовательская – вывод сообщения и закрытие окна. Поскольку используется пользовательский обработчик, то в очереди обрабатываемых сообщений нужно указать этот тип сообщений. Для этого используется макрокоманда ON_COMMAND (IDOK, <ИмяПользовательскогоОбработчика>).

Пусть здесь используется обработчик с прототипом afx_msg void OnButtonOK()

```
afx_msg void MY_DIALOG::OnButtonOK( )
{
    MessageBox( "OnOK","OnButtonOK" );
    EndDialog ( 0 );
};  .
```

Макрокоманда включения имеет вид ON_COMMAND (IDOK, OnButtonOK);

- событие “нажатие кнопки CANCEL” вызывает сообщение WM_COMMAND. Реакция стандартная – закрытие окна. Используется стандартный обработчик и не обязательно указывать явно этот тип сообщений в очереди обрабатываемых сообщений.

3. Специфицировать состав событий-сообщений при работе с главным окном и реакций приложения на эти сообщения:

- события “запуск приложения с перерисовкой окна”, “изменение размеров окна”, “перекрытие окна” и др., приводят к посылке сообщений типа WM_PAINT. Реакция пользовательская – здесь никаких действий не производится, но они могут быть добавлены в дальнейшем. Поскольку здесь используется пользовательский обработчик, то в очереди обрабатываемых сообщений главного окна необходимо явно указать этот тип сообщений макрокомандой включения ON_WM_PAINT();

- событие “нажатие левой клавиши мыши” вызывает сообщение WM_LBUTTONDOWN. Реакция на сообщения этого типа пользовательская – открытие (визуализация) диалогового окна с передачей ему управления. Макрокоманда включения чувствительности окна к сообщениям этого типа ON_WM_LBUTTONDOWN(). Макрокоманде ON_WM_LBUTTONDOWN() соответствует обработчик с прототипом afx_msg void OnLButtonDown(UINT flags, CPoint loc)

```
afx_msg void WINDOW::OnLButtonDown(UINT flags, CPoint loc)
{
    CClientDC dc(this);
    MY_DIALOG TheDialog( (LPTSTR) IDD_DIALOG1, this)  ;
    TheDialog.DoModal();
};   .
```

4. Спроектировать классы приложения. Здесь используется три пользовательских класса. Это класс APPLICATION для создания экземпляра приложения; класс WINDOW для создания объекта – главное окно приложения; класс MY_DIALOG для создания объекта – диалоговое окно. Диаграмма классов, отображающая порядок наследования основных классов, используемых при создании приложения, аналогична диаграмме классов предыдущего примера (рисунок 45).

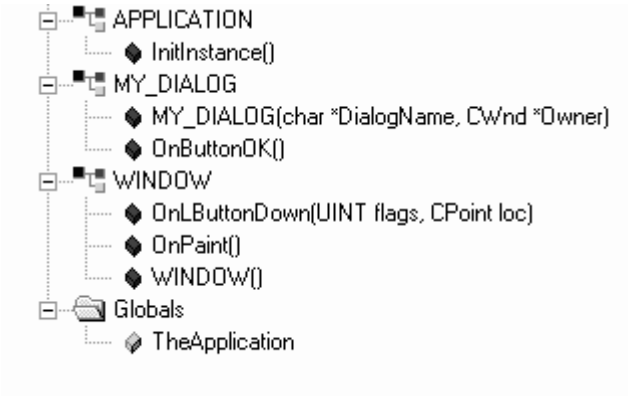


Рисунок 45 - Состав классов

5. Спроектировать модульную структуру приложения. Здесь для представления программной части (за исключением ресурсов) может использоваться один модуль, например, main.cpp.

4 Диалоговое окно в качестве главного окна

ПРИМЕР. *Создать MFC-приложение на базе ТКП с оконным интерфейсом из диалогового окна в качестве главного. Диалоговое окно содержит системное меню и кнопку "Справка", при нажатии на которую выводится справочная информация о приложении. Завершение работы приложения производится соответствующей кнопкой системного меню диалогового окна*.

Приложение создается на базе типового каркаса MFC-приложения в Visual Studio C++. Для создания самого ресурса – диалогового окна используется встроенный редактор ресурсов. ПОРЯДОК (схема) проектирования приложения представлен ниже.

1. Спроектировать интерфейс приложения. Определить состав элементов управления окна и их вид. При необходимости описать окна и ЭУ на соответствующем языке описания ресурсов. Здесь интерфейсные формы включают, как показано ниже (рисунок 46), диалоговое окно, выводимое при запуске приложения в роли главного. Окно содержит кнопку "Справка", при нажатии которой выводится сообщение о приложении. При закрытии окна сообщения управление возвращается диалоговому окну.

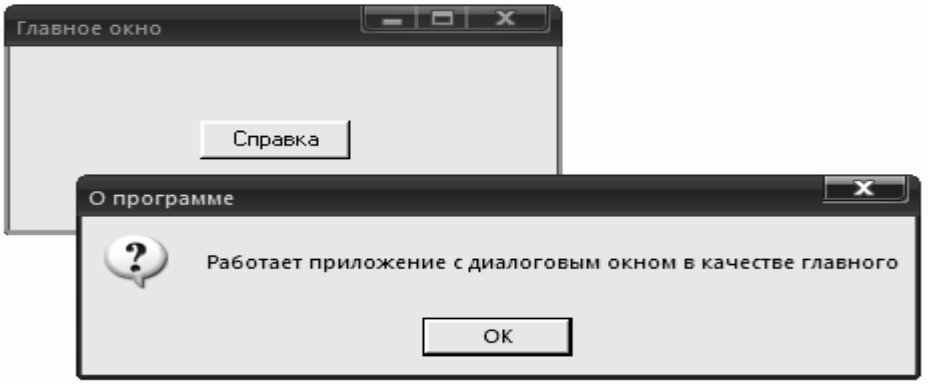


Рисунок 46 - Вид интерфейса

2. Специфицировать состав событий-сообщений при работе с диалоговым (главным) окном и реакций приложения на эти сообщения:

- событие “нажатие кнопки Справка” вызывает сообщение WM_COMMAND. Реакция пользовательская – вывод сообщения на фоне окна. Поскольку используется пользовательский обработчик, то в очереди обрабатываемых сообщений нужно указать этот тип сообщений - в очередь сообщений вставляется макрокоманда, например, ON_COMMAND (IDOK, OnOK). Это означает, что сообщению WM_COMMAND соответствует обработчик с прототипом afx_msg void OnOK().

3. Спроектировать классы приложения. Здесь используются классы: класс class APPLICATION: public CWinApp для создания экземпляра приложения, класс class MY_DIALOG: public CDialog для создания объекта – диалоговое окно.

4. Спроектировать модульную структуру приложения. Здесь для представления программной части (за исключением ресурсов) используется один модуль, например, main.cpp. Код MFC-приложения с диалоговым окном в качестве главного окна представлен ниже

```
#include <afxwin.h>
#include <iostream>
#include "resource.h"
using namespace std;

//////////////////////////////////////////////////////
class MY_DIALOG : public CDialog
{
public:
    MY_DIALOG(char *DialogName, CWnd *Owner): CDialog(DialogName, Owner) { };
    DECLARE_MESSAGE_MAP()
};

BEGIN_MESSAGE_MAP(MY_DIALOG, CDialog)
END_MESSAGE_MAP()

//////////////////////////////////////////////////////
class APPLICATION:public CWinApp
{
public:
        BOOL InitInstance();
};

BOOL APPLICATION::InitInstance()
{
        MY_DIALOG TheDialog((LPTSTR)IDD_DIALOG1,NULL);
        TheDialog.DoModal();
        return TRUE;
}

//////////////////////////////////
APPLICATION  App;  .
```

Диалоговое окно с окошком редактирования. ПРИМЕР. Создать MFC-приложение на базе ТКП для многократного ввода числовых значений и их вывода с противоположным знаком. ПОРЯДОК (схема) проектирования приложения представлена ниже.

1. Спроектировать интерфейс приложения. Определить сценарий работы приложения, состав элементов управления окна и их вид. Интерфейсная форма показана ниже (рисунок 47).

Здесь диалоговое окно выводится при запуске приложения в роли главного.

Окно содержит кнопку “Ввести и вычислить”, поле для ввода числа и поле для вывода результата. Закрытие приложения выполняется кнопкой системного меню.

2. Сценарий работы приложения описан ниже.

В поле ввода диалогового окна задается число, а ввод его в приложение осуществляется по нажатию клавиши Enter или кнопки окна “Ввести и вычислить” с одновременным отображением результата преобразования введенного значения в поле Результат.

Выход из приложения осуществляется по нажатию клавиши Esc или кнопки завершения системного меню диалогового окна.

Соответственно для обработки сообщений, связанных с кнопкой “Ввести и вычислить”, используется пользовательский обработчики сообщений, выполняющий ввод значения из поля ввода в виде строки, преобразование строки в число, выполнение необходимых действий над числом, преобразование числа в строку и вывод строки как результата.

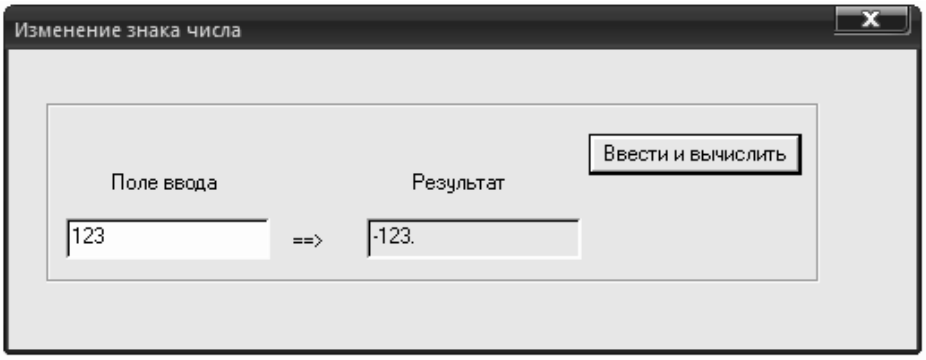


Рисунок 47 - Вид интерфейса

3. Специфицировать состав событий-сообщений при работе с диалоговым (главным) окном и реакций приложения на эти сообщения:

- событие нажатие кнопки “Ввести и вычислить” вызывает сообщение WM_COMMAND.

Реакция пользовательская – считывание данного из окна ввода и его вывод после преобразования в окне вывода.

Поскольку используется пользовательский обработчик, то в очередь сообщений вставляется макрокоманда, например, ON_COMMAND (IDOK, OnOK). Здесь обработчик с прототипом afx_msg void OnOK() обрабатывает сообщения нажатия кнопки “Ввести и вычислить”;

- событие запуска и активизации диалогового окна вызывает сообщение инициализации WM_INITDIALOG.

Реакция на это сообщение – вывод в окне ввода текста приглашения: "Введите число". Прототип обработчика: BOOL OnInitDialog().

4. Спроектировать классы приложения. Здесь используются классы: класс

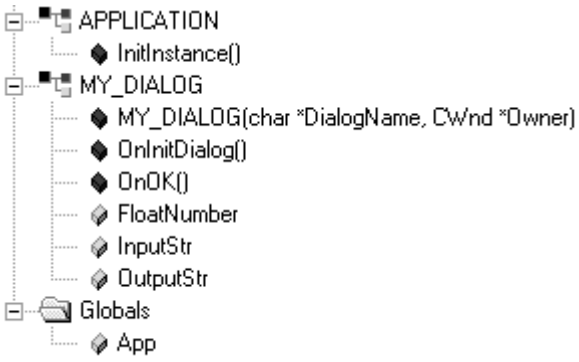


Рисунок 48 - Состав классов

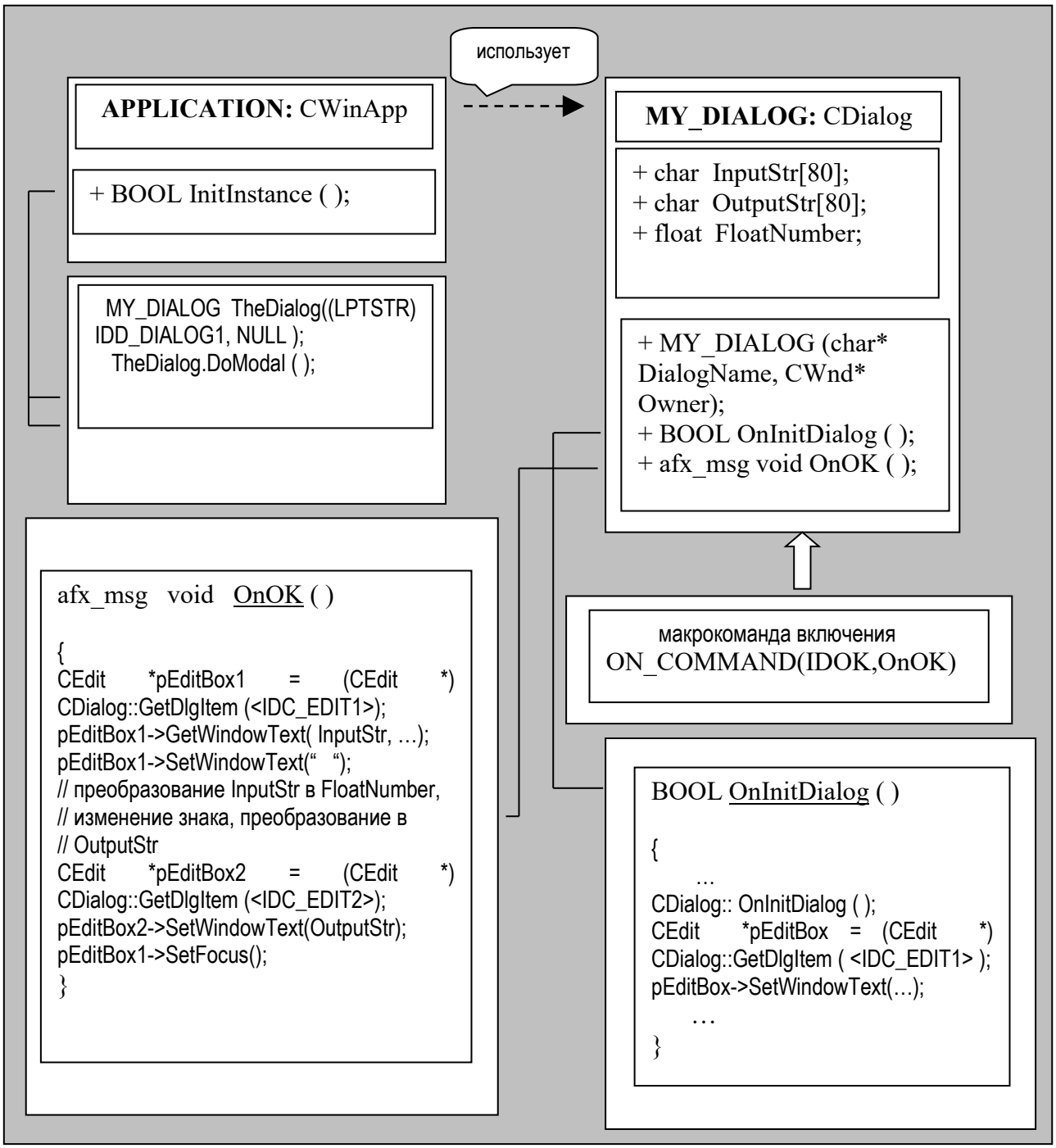


Рисунок 49 - Диаграмма классов

class APPLICATION: public CWinApp

для создания экземпляра приложения; класс

class MY_DIALOG: public CDialog

для создания объекта – диалоговое окно. Состав классов представлен на рисуне 48.

Диаграммы указанных классов, основные члены, методы классов представлены на рисунке 49. Кроме этого неявно используются классы для создания экземпляров объектов элементов управления типа: – окно редактирования, статичное окно, рамка (группирующий элемент управления), командная кнопка.

5. Спроектировать модульную структуру приложения. Здесь для представления программной части (за исключением ресурсов) используется один модуль.

Лекция № 11 Разработка интерфейсов на базе библиотеки MFC

1 Общие сведения о списках. Класс CListBox

Назначение, общее описание. Список – это элемент управления, используемый в составе окна. Соответственно все адресуемые списку сообщения являются сообщениями его окна и включаются в его очередь сообщений, а “привязка” к конкретному списку производится с помощью параметра сообщения, содержащего ID_Списка.

Список хранит множество элементов, строк и поддерживает предопределенный набор действий над ними, а каждая строка имеет идентифицирующий номер – индекс, исчисляемый от нуля. Список хранимых элементов может, например, включать имена файлов, фамилий сотрудников и т.п. Пользователь может просматривать указанный список и делать свой выбор.

В списках с единичным выбором (single-selection list box) пользователь может выбрать только один элемент. В списках с множественным выбором (multiple-selection list box) может выделяться диапазон элементов. Результат выбора подсвечивается, а сам список посылает уведомляющее сообщение (notification message) родительскому окну.

В файле ресурсов каждый список описывается командой LISTBOX.

Функциональность элемента управления Windows типа список (list box) в MFC непосредственно поддерживается классом CListBox, точнее объектами класса CListBox (фрагмент диаграммы классов представлен ниже - рисунок 50).

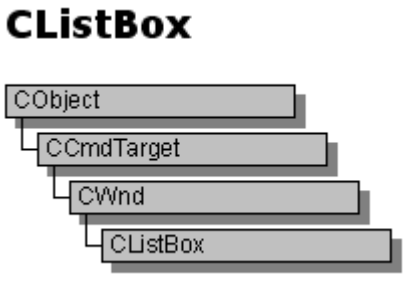


Рисунок 50 - Иерархия классов CListBox

Списки, как и другие элементы управления, могут создаваться как ресурс вне приложения на основе соответствующего оконного шаблона (dialog template) либо непосредственно программным путем – в самом приложении.

При работе со списками следует подключить #include <afxwin.h>

Состояние списка может меняться: - под воздействием действий пользователя, событий; - программно. Для этого список может как посылать, инициировать сообщения, так и принимать, выполнять команды. В первом случае действия пользователя вызывают информирующие сообщения, которые могут быть приняты обработчиком списка (его окна) и соответствующим образом обработаны, учтены. Во втором случае используются команды списка (методы и операторы класса CListBox), позволяющие влиять на состояние списка программно, например, из его обработчика.

Стили. Основные разновидности списка: список с единичным выбором; список с множественным выбором; список только для отображения текста и т.д. Полный перечень стилей списков, команды управления списком приведены в таблицах 27, 28 ниже.

Создание и удаление. Списки могут создаваться на основе соответствующего оконного шаблона (dialog template) либо непосредственно программным путем. В первом случае список создается визуально в редакторе ресурсов либо описывается в текстовом редакторе. В обоих случаях далее создается объект класса CListBox соответствующим конструктором CListBox.

При непосредственном создании вначале создается локальный объект, например, как CListBox *МойСписок* или динамический объект, например, как CListBox* *Указатель-МойСписок* = new CListBox; .

Затем вызывается метод Create для создания самого элемента управления и “прикрепления” его к этому объекту. Метод может вызываться в конструкторе объекта.

В случае создания списка на базе оконного шаблона описывается переменная в классе родительского диалогового окна, затем используется средство DDX_Control (DoDataExchange) для связывания переменной и элемента управления. Указанное выполняется мастером ClassWizard автоматически каждый раз, когда в класс диалогового окна добавляется такая переменная.

Если список создан как ресурс вне приложения на основе соответствующего оконного шаблона (dialog template), то соответствующий CListBox-объект автоматически разрушается, когда пользователь закрывает родительское диалоговое окно.

Если список создан программным путем внутри окна, то о его разрушении должен позаботиться программист в самом приложении. Например, если CListBox-объект создан методом new, то необходимо вызвать метод delete для его разрушения, когда пользователь закрывает родительское диалоговое окно.

Обработка сообщений. Для обработки уведомляющего сообщения (notification message), посылаемого родительскому окну (как правило, объекту класса, производного от CDialog) списком, необходимо ввести в карту сообщений окна макрокоманду со стандартной ON_COMMAND-сигнатурой вида

ON_Notification(*ID_Списка*, *ИмяОбработчикаСообщения*) ,

где ID специфицирует список-источник сообщения и добавить обработчик сообщения с прототипом

afx_msg void *ИмяОбработчикаСообщения* ( ) .

Посылаемые сообщения. Список сообщений можно получить в MSDN, а информацию о сообщении путем поиска, например, по образцу LBN_, LBN_DBLCLK. Ниже дана общая характеристика типовым событиям и соответствующим уведомляющим сообщениям (notification message):

- сообщение LBN_DBLCLK посылается, если пользователь дважды ”щелкает” на строке списка для ее выбора (при условии, что стиль списка LBS_NOTIFY);
- сообщение LBN_ERRSPACE посылается, если не хватает памяти для обслуживаниия запроса;
- сообщение LBN_KILLFOCUS посылается, если список теряет фокус ввода;
- сообщение LBN_SELCANCEL посылается, если выбор отменен (при условии, что стиль списка LBS_NOTIFY);
- сообщение LBN_SELCHANGE посылается, если выбор в списке меняется (при условии, что стиль списка LBS_NOTIFY). Оно не посылается, если выбор менялся как результат выполнения команды-метода CListBox::SetCurSel. Для списков с множественным выбором сообщение посылается при нажатии клавиш-стрелок, даже если выбор пользователя не меняется;
- сообщение LBN_SETFOCUS посылается, если список получает фокус ввода.

Члены класса. Основные методы класса подразделяются на: - методы-конструкторы; - методы инициализации; - общие методы; - методы списка с единичным выбором; - методы списка с множественным выбором; - строковые методы. Прототипы наиболее часто используемых методов приведены ниже.

Общая схема создания списка. Включает следующий набор действий:

1) определение вида, типа списка, его свойств (стиля);

2) описание всех событий, связанных со списком, соответствующих сообщений, описание реакции на них (обработчиков), описание макрокоманд обеспечения чувствительности приложения (окна) к сообщениям списка. Т.е. выполнение действий, например, в следующей последовательности:

тип сообщения LBN_< *НазваниеСообщения* > ->

обработчик < *ИмяОбработчикаСообщения* > ->

выполняемые действия как < *СпецификацияОбработчикаСообщений* > ->

макрокоманда включения как ON_LBN_<*НазваниеСообщения*> (<*ID_Списка*>, <*ИмяОбработчикаСообщения* >) ;

3) описание списка в составе окна-подложки (диалогового окна) в файле ресурсов (командой описания LISTBOX), например, с помощью редактора ресурсов, и установка его свойств;

4) настройка класса окна (например, *ДИАЛОГОВОЕ_ОКНО*: CDialog), содержащего список, на работу с ним:

- в описание класса окна необходимо вставить прототипы всех функций-обработчиков сообщений списка (например, afx_msg void On< *ИмяОбработчикаСообщения* > ( ));
- необходимо описать каждый обработчик сообщений списка на основе <*СпецификацииОбработчикаСообщений* >;
- включить чувствительность списка к соответствующим сообщениям - в описание очереди сообщений окна со списком включить необходимые макрокоманды (например, ON_LBN_DBLCLK (< *ID_Списка* >, < *ИмяОбработчикаСообщения* >));
- при необходимости, выполнить инициализирующие действия относительно списка в функции OnInitDialog( ) соответствующего окна-подложки (например, задать начальное содержимое списка).

Общая схема работы со списком. Для работы со списком (это общее правило работы с элементами управления) необходимо получить на него указатель, используя функцию

CWnd *CWnd::GetDlgItem (int *ID_Списка*) const .

Через указатель далее можно вызывать необходимые методы, посылать списку команды. Например

CListBox *pCListBox = (CListBox *) GetDlgItem (ID_CListBox) ,

где pCListBox (или lbptrCListBox) и есть указатель, через который вызываются все методы списка, например,

```
int  i = pCListBox -> GetCount( );
int  i = pCListBox -> GetCurSel( ).
```

Для инициализации содержимого списка в описании класса его родительского окна используется метод

OnInitDialog ( )

Общая схема использования

```
BOOL ДИАЛОГОВОЕ_ОКНО::OnInitDialog( )
{
  CDialog::OnInitDialog ( );
  Получить_указатель_на_список (например, CListBox *pListBox = (CListBox *) GetDlgItem( ID_Списка ) );
  Инициализировать_список (например, pListBox -> AddString( < ОднаСтрокаСписка > );
  ……
  return TRUE;
  }; .
```

Стили элемента управления список, команды приведены ниже в таблице 27, 28.

Таблица 27 - Стили элемента управления список

Стиль	Описание
LBS_DISABLENOSCROLL	- вертикальная полоса прокрутки не отображается, если в списке недостаточно элементов, чтоб их “прокручивать”
LBS_EXTENDEDSEL	- позволяет пользователю выбирать сразу несколько элементов, строк, используя клавишу SHIFT, “мышь” или горячие клавиши
LBS_MULTICOLUMN	- многоколонковый список с возможностью горизонтальной прокрутки его содержимого (ширина колонки устанавливается с помощью сообщения LB_SETCOLUMNWIDTH)
LBS_MULTIPLESEL	- допускается выбор произвольного количества элементов, строк списка. При этом выбранная строка меняется каждый раз, когда пользователь выполняет “щелчок” на строке

LBS_NOSEL	- пользователь может просматривать список, но не может выбирать строки
LBS_NOTIFY	- при одинарном или двойном “щелчке” на строке посылается уведомляющее сообщение (LBN_DBLCLK, LBN_SELCANCEL, LBN_SELCHANGE)
LBS_SORT	- строки сортируются в алфавитном порядке, при одинарном или двойном “щелчке” на строке посылается уведомляющее сообщение
LBS_STANDARD	- строки сортируются в алфавитном порядке. Родительское окно получает сообщение ввода каждый раз при одинарном или двойном “щелчке” на строке. Сам список - с рамкой и полосой прокрутки
WS_TABSTOP	- используется в диалоговых окнах, позволяя пользователю перемещаться между элементами управления этого стиля с помощью клавиши tab

Таблица 28 - Команды (методы) управления списком

CListBox	- конструктор, создает объект класса CListBox
Create	- инициализатор, создает элемент управления Windows типа list box и прикрепляет его к объекту класса CListBox
InitStorage	- распределяет память для элементов списка
GetCount	- общий метод, возвращает число строк в списке
GetTopIndex	- общий метод, возвращает номер первой видимой строки
GetItemData	- общий метод, возвращает 32-битовое значение элемента с
SetItemData	- общий метод, устанавливает 32-битовое значение, ассоциируемое с элементом списка
GetSel	- общий метод, возвращает состояние выбора элемента
GetText	- общий метод, читает элемент списка в буфер
GetTextLen	- общий метод, возвращает длину в байтах элемента списка
GetCurSel	- возвращает номер выбранной строки
SetCurSel	- для списка с одиночным выбором выделяет или снимает выделение строки
SetSel	- для списка с множественным выбором выделяет или снимает выделение строк
GetSelCount	- для списка с множественным выбором возвращает число выделенных строк
AddString	- строковый метод, добавляет новую строку в список

DeleteString	- строковый метод, удаляет строку из списка
InsertString	- строковый метод, вставляет новую строку в указанное место списка
FindString	- строковый метод, обеспечивает поиск строки
FindStringExact	- строковый метод, обеспечивает поиск первой строки, которая совпадает с образцом
SelectString	- строковый метод, ищет и выделяет строку

Диалоговое окно со списком в качестве главного окна. ПРИМЕР. *Создать MFC-приложение на базе ТКП для ведения списка записей - фамилий. Исходный, а затем и модифицированный список фамилий должен отображаться в диалоговом окне в соответствующем списковом элементе управления (класса CListBox). Исходный список создается в момент инициализации диалогового окна последовательным добавлением к списку заранее определенных фамилий.*

Пользователь может осуществить выбор (выделение) фамилии одинарным либо двойным щелчком "мыши". Выбранную (выделенную) фамилию здесь можно: - удалить; - использовать для создания и добавления к списку новой фамилии. Каждое из трех действий (выбор, удаление, добавление) должно сопровождаться выводом сообщения о его выполнении и требованием на подтверждение.

Добавление фамилии здесь производится в упрощенном варианте и состоит во вводе в список копии уже содержащейся фамилии, выбранной одинарным или двойным щелчком, с добавлением к ней числового номера (например, Иванов1, Иванов5, …).

Для разработки приложения необходимо спроектировать следующие компоненты.

1. Разработать сценарии работы приложения. Здесь в полях списка диалогового окна выводится перечень строк – фамилий.

Фамилию можно выбрать щелчком мыши (этому событию соответствует сообщение LBN_CHANGESEL), двойным щелчком (этому событию соответствует сообщение LBN_DBLCLK). Выбор пользователя должен сопровождаться его же подтверждением.

Выделенную фамилию можно удалить нажатием кнопки Удалить ФИО. Для этого используется метод int DeleteString(UINT *nIndex*) .

Новую фамилию можно добавить нажатием кнопки Добавить ФИО. Здесь добавление фамилии будет состоять во вводе копии уже содержащейся фамилии (выбранной одинарным или двойным щелчком) с добавлением к ней соответствующего числового номера. Для добавления новой фамилии к списку используется метод int AddString(LPCSTR *lpszStr*) .

Каждое из перечисленных выше действий сопровождается сообщением о его выполнении и требованием подтверждения.

Соответственно в приложении понадобятся обработчики: - сообщений типа LBN_DBLCLK для вывода результата выбора; - обработчики сообщений нажатия кнопок Удалить ФИО и Добавить ФИО, приводящие к соответствующей коррекции списка.

2. Определить состав графического интерфейса приложения, стили элементов интерфейса. Здесь в качестве главного окна будет использоваться диалоговое окно (поддерживаемое классом *MY_DIALOG)* со списком и двумя кнопками управления списком. Кроме этого используется статичный элемент – рамка с названием "Управление списком". Примерный вариант интерфейса представлен ниже (рисунки 51-53). Там же пока-

зан вид интерфейса – вывод предупреждения - в случае попытки выполнения команды без выбора фамилии.

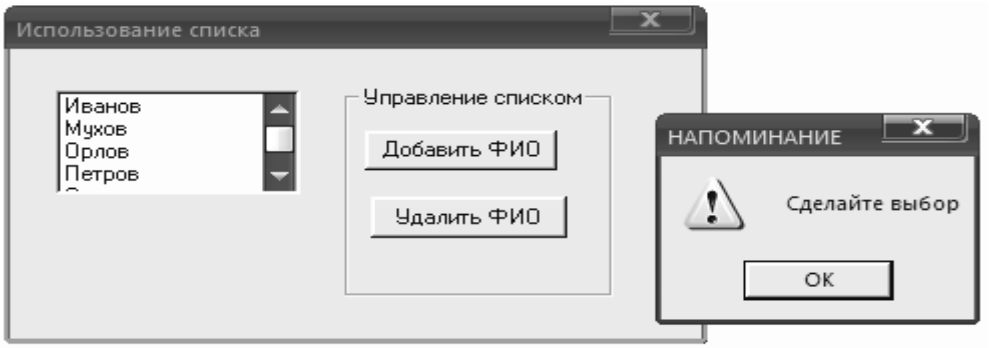


Рисунок 51 - Интерфейс приложения

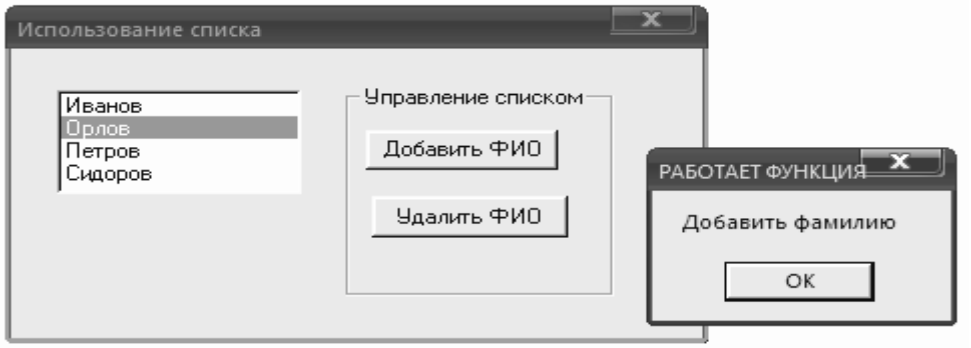


Рисунок 52 - Интерфейс приложения. Подтверждение команды

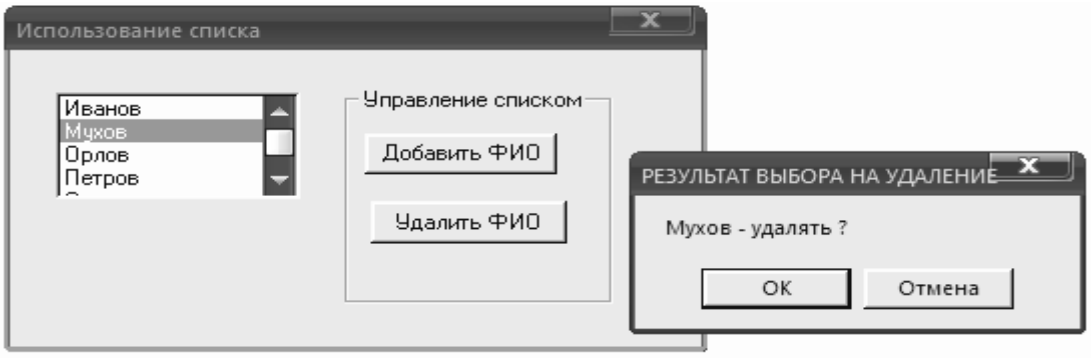


Рисунок 53 - Интерфейс приложения. Подтверждение выполнения

3. Специфицировать состав событий-сообщений при работе с диалоговым (главным) окном и реакций приложения на эти сообщения: - событие “двойной щелчок на элементе списка” вызывает сообщение LBN_DBLCLK. Чувствительность окна, содержащего этот список, задается макрокомандой ON_LBN_DBLCLK(<*ID_Списка*>, <*ИмяОбработчика-Сообщения*>).

Реакция на сообщение – вывод выбранной фамилии. Поскольку используется пользовательский обработчик, то в очереди обрабатываемых сообщений нужно указать этот тип сообщений - в очередь сообщений вставляется макрокоманда, например, ON_LBN_DBLCLK(<*ID_Списка*>, *ToDisplaySelectedFIO*). Это означает, что сообщению

соответствует обработчик с прототипом afx_msg void *MY_DIALOG*:: *ToDisplaySelectedFIO* ( );

- событие “нажатие кнопки Добавить ФИО” вызывает сообщение WM_COMMAND. Реакция на сообщение – добавление фамилии к списку. В очередь сообщений вставляется макрокоманда, например, ON_COMMAND (<*ID_ КнопкиДобавитьФИО*>, *ToAddFIO*). Это означает, что сообщению соответствует обработчик с прототипом afx_msg void *MY_DIALOG*:: *ToAddFIO* ( );

- событие “нажатие кнопки Удалить ФИО” вызывает сообщение WM_COMMAND. Реакция – удаление фамилии. В очередь сообщений вставляется макрокоманда, например, ON_COMMAND (<*ID_КнопкиУдалитьФИО*>, *ToDeleteFIO*). Это означает, что сообщению соответствует обработчик с прототипом afx_msg void *MY_DIALOG*:: *ToDeleteFIO* ( ).

4. Спроектировать классы приложения. Здесь используются следующие классы: - класс myAPPLICATION: public CWinApp для создания экземпляра приложения; - класс myDIALOG: public CDialog для создания объекта – диалоговое окно.

5. Спроектировать модульную структуру приложения. Здесь для представления программной части (за исключением ресурсов) используется один модуль, например, main.cpp. Код приложения представлен ниже.

```
#include <afxwin.h>
#include <iostream>
#include "resource.h"
using namespace std;

class myDIALOG : public CDialog
{
public:
        myDIALOG(char *DialogName, CWnd *Owner): CDialog(DialogName, Owner)
        {       };
        BOOL OnInitDialog();
        afx_msg void OnToDisplaySelectedFIO();
        afx_msg void OnToAddFIO();
        afx_msg void OnToDeleteFIO();
        DECLARE_MESSAGE_MAP()
};

BEGIN_MESSAGE_MAP(myDIALOG, CDialog)
        ON_LBN_DBLCLK(IDC_LIST1, OnToDisplaySelectedFIO)
        ON_COMMAND(IDC_BUTTON1, OnToAddFIO)
        ON_COMMAND(IDC_BUTTON2, OnToDeleteFIO)
END_MESSAGE_MAP()

afx_msg void myDIALOG::OnToDisplaySelectedFIO()
{
        char Str[80];
        MessageBox("Выбор двойным щелчком","РАБОТАЕТ ФУНКЦИЯ");
        CListBox *PListBox = (CListBox *) GetDlgItem(IDC_LIST1);
```

```
        int i = PListBox ->GetCurSel();
        PListBox ->GetText(i,Str);
        strcat(Str," - ваш выбор");
        MessageBox(Str,"РЕЗУЛЬТАТ ВЫБОРА");
};

afx_msg void myDIALOG::OnToAddFIO()
{
        static int j=0;
        char Str[80];char Str1[80];char Str2[80];
        j++;
        MessageBox("Добавить фамилию","РАБОТАЕТ ФУНКЦИЯ");
        CListBox *PListBox = (CListBox *) GetDlgItem(IDC_LIST1);
        int i = PListBox ->GetCurSel();
        if (i == LB_ERR )
        {
              MessageBox("Сделайте выбор","НАПОМИНАНИЕ",MB_OK |
                                    MB_ICONWARNING);
         }
        else
        {
             PListBox ->GetText(i,Str);
             strcpy(Str1,Str);
             strcat(Str1," - ваш выбор");
             MessageBox(Str1,"РЕЗУЛЬТАТ ВЫБОРА На ДОБАВЛЕНИЕ");
             wsprintf(Str2,"%d",j);
             strcat(Str,Str2);
             PListBox -> AddString(Str);
        };
};

afx_msg void myDIALOG::OnToDeleteFIO()
{
        char Str[80];
        MessageBox("Удалить фамилию","РАБОТАЕТ ФУНКЦИЯ");
        CListBox *PListBox = (CListBox *) GetDlgItem(IDC_LIST1);
        int i = PListBox ->GetCurSel();
        if (i == LB_ERR )
        {
            MessageBox("Выбирайте!","НАПОМИНАНИЕ", MB_OK|MB_ICONWARNING);
        }
        else
        {
                i = PListBox ->GetCurSel();
                PListBox ->GetText(i,Str);
```

```
                strcat(Str," - удалять ?");
                if (MessageBox(Str,"РЕЗУЛЬТАТ ВЫБОРА ", MB_OKCANCEL) == IDOK)
                {
                        PListBox ->DeleteString(i);
                };
        };
};

BOOL myDIALOG::OnInitDialog()
{
        CDialog::OnInitDialog();
        CListBox *PListBox = (CListBox *) GetDlgItem(IDC_LIST1);
        PListBox -> AddString("Иванов");
        PListBox -> AddString("Петров");
         return TRUE;
};

class myAPPLICATION:public CWinApp
{
public:
        BOOL InitInstance();
};

BOOL myAPPLICATION::InitInstance()
{
        myDIALOG TheDialog((LPTSTR)IDD_DIALOG1,NULL);
        TheDialog.DoModal();
        return TRUE;
}

myAPPLICATION  ThisApplication;  .
```

2 Использование пользовательского меню

Меню представляет собой набор пунктов и может рассматриваться как определенным образом организованное множество (система) кнопок, где каждому пункту меню соответствует кнопка. Кнопки-пункты объединяются в подменю. Пользователь может перемещаться от меню верхнего уровня к подменю нижних уровней. Кнопки, пункты меню бывают разных типов. Это, например, пункты типа: - конечный пункт MENUITEM; - “всплывающий” пункт POPUP (pop-up в Visual Studio C++); - разделитель пунктов MENUBARBREAK (separator в Visual Studio C++).

При этом кнопки меню типа POPUP автоматически при нажатии обеспечивают переход к подменю более низкого уровня - другим кнопкам меню в соответствии с его структурой. Они поддерживают навигацию пользователя в системе меню. А конечные кнопки типа MENUITEM работают как командные кнопки (например, как кнопки диалоговых окон

OK, Cancel и т.д.). Их нажатие приводит к генерации сообщения типа WM_COMMAND, чувствительность к которому включается макрокомандой

ON_COMMAND(< *ID_пункта_меню* >, < *ИМЯ_обработчика_пункта_меню* >) ,

и, в конечном итоге, приводит к запуску соответствующего выбранному пункту меню обработчика.

Стили меню и пунктов меню приведены ниже в таблице 29.

Общая схема создания меню включает следующий набор действий.

1. Проектирование меню.

1.1. Описание общих свойств пользовательского меню. Например, описание названия меню, определение необходимости динамической загрузки меню или постоянного размещения в памяти и т.п.

1.2. Разработка структуры пользовательского меню, состава пуктов и их соподчинения. Для каждого пункта меню следует определить:

- тип пункта (MENUITEM, POPUP, MENUBARBREAK);
- свойства пункта, включая название пункта (caption) с отметкой при необходимости предваряющим символом & (например, F&ile) наличия горячей клавиши. Определить необходимость автоматической отметки выбранного пользователем пункта (свойство CHECKED), отсутствия подсветки пункта в случае отсутствия выбора (свойство GRAYED), временной не активности, не доступности пункта для выбора (свойство INACTIVE);
- желаемую реакцию приложения. Для выбранного конечного пункта меню это набор действий, описаваемых в обработчике. Для выбранного “всплывающего” пункта меню это активизируемое подменю и т.д.

2. Описание меню в файле ресурсов, например, с помо щью редактора ресурсов аналогично тому как, например, описываются ресурсы типа “окно”.

3. Подключение меню к окну приложения. Например, через соответствующий параметр метода create.

4. Настройка класса окна приложения на работу с пользовательским меню.

4.1. В описание класса окна следует вставить прототипы (например, afx_msg void On<ИмяОбработчика> ( )) функций-обработчиков сообщений для всех событий типа “выбор конечного пункта меню”.

4.2. Описать функции-обработчики сообщений события “выбор конечного пункта меню”.

4.3. Включить чувствительность окна к сообщениям события “выбор конечного пункта меню” с помощью размещения в карте окна макрокоманд типа ON_COMMAND (<ID_ПунктаМеню>, < ИмяОбработчика >).

Таблица 29 - Стили меню, пунктов меню

Стиль	**Описание**
CHECKED	- опция пункта, требует размещения рядом с выбранным пользователем пунктом меню отметки выбора (для пунктов меню верхнего уровня не используется)
GRAYED	- опция пункта, требует пометки пункта меню как не активного (серым, “бледным” цветом). Такой пункт не может быть выбран пользователем
HELP	- опция пункта, определяет, что пункт меню может быть связан с командой вызова помощи (применяется только с пунктами

	меню типа MENUITEM)
INACTIVE	- опция пункта, требует, чтобы пункт меню выводился в списке меню, но не мог быть выбран в данных обстоятельствах
MENUBREAK	- опция пункта, определяет свойство аналогичное MENUBARBREAK, но без использования разделительной черты
MENUBARBREAK	- опция пункта, используется для указания пункта меню, выполняющего роль разделителя для других пунктов. В меню верхнего уровня вызывает запись названия нового пункта с новой строки. В выпадающих меню название пункта будет размещено в новом столбце и отделено чертой
OWNERDRAW	- опция пункта, используется для указания, что состоянием и изображением пункта меню, включая выделенное, неактивное и отмеченное состояния, отвечает “владелец” меню
POPUP	- опция пункта, используется для указания типа пункта меню как POPUP. При выборе этого пункта выводится список пунктов подменю
DISCARDABLE	- опция меню, используется для указания, что меню может быть удалено из памяти, если оно больше не используется
FIXED	- опция меню, требует постоянного нахождения меню в памяти
LOADONCALL	- опция меню, используется для указания необходимости загрузки меню каждый раз при обращении к нему

Приложение с пользовательским меню. ПРИМЕР. *Создать приложение на базе ТКП с пользовательским меню. При выборе каждого конечного пункта менюдолжно выводиться подтверждающее сообщение. Завершение приложения производится закрытием главного окна либо выбором первого пункта меню. Предполагается, что визуализация окна с меню должна происходить автоматически каждый раз при запуске приложения.*

Приложение создается на базе типового каркаса MFC-приложения, а для создания ресурса – меню используется встроенный редактор ресурсов как альтернатива описанию меню текстом в файле ресурсов.

Для создания приложения необходимо выполнить следующие действия.

1. Спроектировать приложение.

1.1. Разработать интерфейс приложения. Здесь это главное окно с пользовательским меню как показано ниже (рисунки 54,55).

1.1.1. Определить состав меню, свойства пунктов, их вид. Структура меню представлена на рисунке ниже. Меню включает главное и два подменю:

- меню верхнего уровня из двух пунктов (Пункт_1, Пункт_2);
- подменю Пункта_2 из двух пунктов (Пункт_2.1, Пункт_2.2);
- подменю Пункта_2.2 из одного пункта (Пункт_2.2.1).

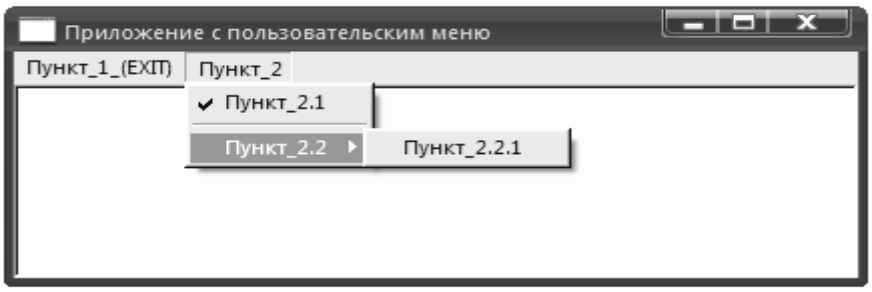


Рисунок 54 - Интерфейс приложения

Соответственно пункты меню Пункт_2, Пункт_2.2 - выпадающие пункты типа Pop-up, содержащие подпункты. А остальные пункты меню - конечные типа MenuItem. Для разделения пунктов меню Пункт_2.1 и 2.2 в подменю используется разделитель (пункт типа Separator). Для отметки выбора пункта меню Пункт_2.1 используется флажок (устанавливается свойство Checked).

1.1.2. Специфицировать состав событий-сообщений при работе с меню и реакций приложения на эти сообщения.

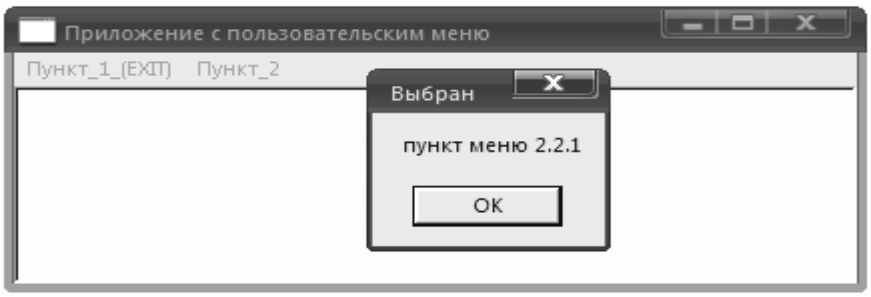


Рисунок 55 - Выбор пункта меню

События-сообщения при работе с меню здесь относятся к типу события “нажатие кнопки выбранного пункта меню мышью или с помощью “горячей” клавиши”, что вызывает сообщение WM_COMMAND.

Поскольку здесь три конечных пункта меню, то будут использоваться обработчики, например, с прототипами

afx_msg void OnMenuItem_1 ( );
afx_msg void *OnMenuItem_2_1* ( );
afx_msg void *OnMenuItem_2_2_1* ( );

и соответствующие макрокоманды включения. По условию выбор каждого конечного пункта меню кроме первого (Пункт_1) сопровождается подтверждением. Выбор первого пункта должен приводить к закрытию окна и приложения.

1.2. Спроектировать классы приложения. Здесь используется типовой каркас с двумя классами: - класс ПРИЛОЖЕНИЕ (class *APPLICATION*: public CWinApp) для создания экземпляра приложения); - класс ОКНО (здесь class *WINDOW*: public CFrameWnd) для создания объекта – главное окно приложения. Состав классов иллюстрируется ниже (рисунок 56).

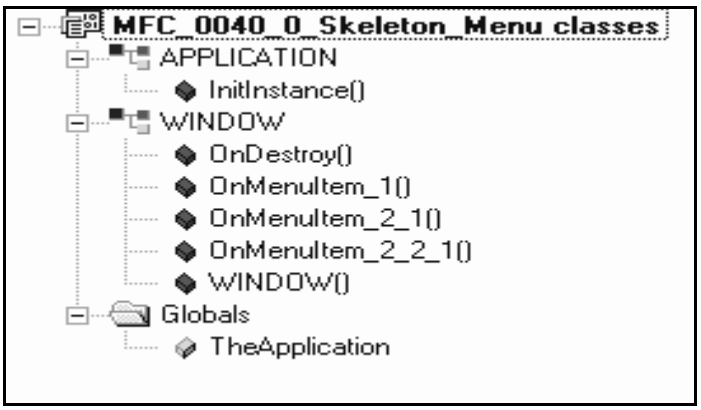


Рисунок 56 - Состав классов приложения

1.3. Спроектировать модульную структуру приложения. Код приложения приведен ниже.

```
#include <afxwin.h>
#include <iostream>
#include "resource.h"
using namespace std;

class WINDOW: public CFrameWnd
{
public:
        WINDOW();
        afx_msg void OnMenuItem_1 ();
        afx_msg void OnMenuItem_2_1 ();
        afx_msg void OnMenuItem_2_2_1 ();
        //afx_msg void OnDestroy ( );
        DECLARE_MESSAGE_MAP()
};

WINDOW::WINDOW()
{
        Create(NULL, "Приложение с пользовательским меню",
        WS_OVERLAPPEDWINDOW, rectDefault,
        NULL, (LPTSTR) 101);
        // или  NULL, (LPTSTR) IDR_MENU1);
}

afx_msg void WINDOW:: OnMenuItem_1 ( )
{
         MessageBox("пункт меню 1 "," Выбран  ");
        SendMessage(WM_CLOSE);
};

afx_msg void WINDOW:: OnMenuItem_2_2_1 ( )
```

```
{
        MessageBox("пункт меню 2.2.1 "," Выбран   ");
};

afx_msg void WINDOW:: OnMenuItem_2_1  ( )
{
        MessageBox("пункт меню 2.1 "," Выбран   ");
};

BEGIN_MESSAGE_MAP(WINDOW,CFrameWnd)
         ON_COMMAND(IDM_MenuItem_1, OnMenuItem_1)
         ON_COMMAND(IDM_MenuItem_2_1, OnMenuItem_2_1)
         ON_COMMAND(IDM_MenuItem_2_2_1, OnMenuItem_2_2_1)
END_MESSAGE_MAP()

class APPLICATION : public CWinApp
{
public:
        BOOL InitInstance();
};

BOOL APPLICATION :: InitInstance()
{
        m_pMainWnd =  new WINDOW;
        m_pMainWnd -> ShowWindow(m_nCmdShow);
        m_pMainWnd -> UpdateWindow();
        return TRUE;
}

APPLICATION  TheApplication; .
```

Приложение с пользовательским меню и ДО. ПРИМЕР. *Создать приложение с пользовательским меню, предназначенное для многократного ввода строк. При выборе пункта меню Enter должно выводиться диалоговое окно для ввода новой строки. При выборе пункта Display должна отображаться последняя введенная строка. Завершение работы приложения производится закрытием главного окна либо выбором пункта меню Exit.*

Для создания приложения необходимо спроектировать приложение.

1. Спроектировать интерфейс приложения. Разработать структуру окон, структуру меню, свойства пунктов меню, их вид. Здесь будут использованы: - главное окно с пользовательским меню; - диалоговое окно для ввода строки; - окна сообщений для вывода строки и сообщений. Интерфейс приложения представлен на рисунках 57-59 ниже.

Рисунок 57 - Интерфейс приложения. Завершение работы

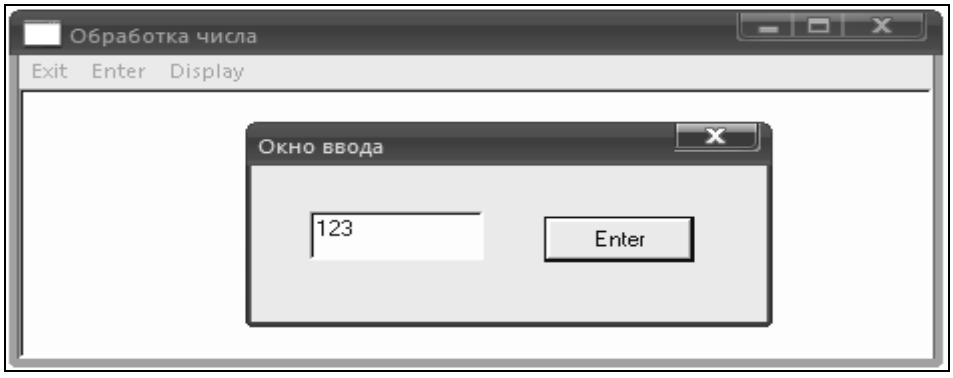


Рисунок 58 - Интерфейс приложения. Ввод строки

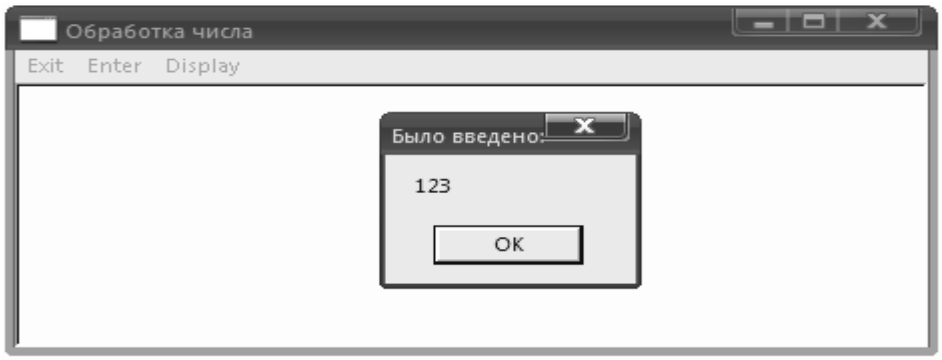


Рисунок 59 - Интерфейс приложения. Вывод строки

2. Специфицировать состав событий-сообщений. При работе с меню это события типа “нажатие кнопки выбранного пункта меню”, приводящие к сообщениям WM_COMMAND. Здесь три конечных пункта меню, соответственно будут использоваться обработчики с прототипами

afx_msg void *OnExit*( );
afx_msg void *OnEnter*( );
afx_msg void *OnDisplay*( );

и соответствующие макрокоманды включения.

При работе с диалоговым окном ввода это события нажатия кнопки Enter, приводящие к сообщениям WM_COMMAND. Прототип обработчика afx_msg void *OnEnter* ( ); .

3. Спроектировать классы приложения. Здесь используются следующие классы: - класс ПРИЛОЖЕНИЕ (class APPLICATION: public CWinApp) для создания экземпляра приложения; - класс ГЛАВНОЕ_ОКНО (class WINDOW: public CFrameWnd) для создания объекта – главное окно приложения; - класс ДИАЛОГОВОЕ_ОКНО (class DIALOG_WINDOW: public CDialog) для создания объекта – окно ввода. Примерный состав классов представлен ниже (рисунок 60).

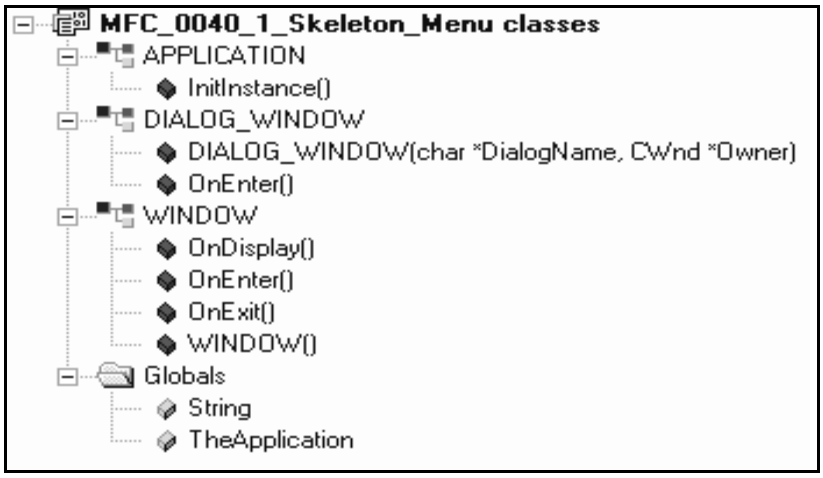


Рисунок 60 - Классы приложения

4. Спроектировать модульную структуру приложения - представлена ниже (рисунок 61). Далее представлены варианты кодов приложения.

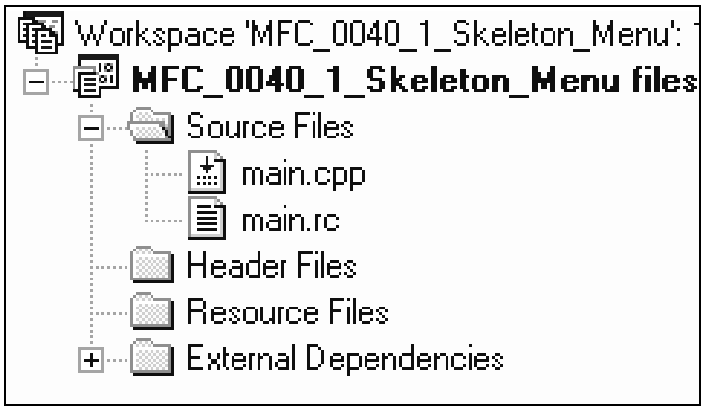


Рисунок 61 - Структура приложения

```
#include <afxwin.h>
#include <iostream>
#include "resource.h"
using namespace std;

char String[80];
class DIALOG_WINDOW : public CDialog
{
public:
        DIALOG_WINDOW(char *DialogName, CWnd *Owner):
```

```
        CDialog(DialogName, Owner){};
        afx_msg void OnEnter();
        DECLARE_MESSAGE_MAP()
};

BEGIN_MESSAGE_MAP(DIALOG_WINDOW, CDialog)
        ON_COMMAND(ID_ENTER, OnEnter)
END_MESSAGE_MAP()

afx_msg void DIALOG_WINDOW::OnEnter()
{
        CEdit *ptr = (CEdit *) GetDlgItem(IDC_EDIT);
        ptr->GetWindowText(String, sizeof String-1);
};

class WINDOW:public CFrameWnd
{
public:
        WINDOW();
        afx_msg void OnExit();
        afx_msg void OnEnter();
        afx_msg void OnDisplay();
        DECLARE_MESSAGE_MAP()
};

WINDOW::WINDOW()
{
        Create(NULL, "Обработка числа", WS_OVERLAPPEDWINDOW, rectDefault,
                NULL, (LPTSTR) IDR_MENU);
}

BEGIN_MESSAGE_MAP(WINDOW, CFrameWnd)
         ON_COMMAND(IDM_EXIT, OnExit)
         ON_COMMAND(IDM_ENTER, OnEnter)
         ON_COMMAND(IDM_DISPLAY, OnDisplay)
END_MESSAGE_MAP()

afx_msg void WINDOW::OnEnter()
{
        DIALOG_WINDOW MyDataEnterDialog((LPTSTR)IDD_DIALOG ,this);
        MyDataEnterDialog.DoModal();
};

afx_msg void WINDOW::OnDisplay()
{
```

```
        MessageBox(String,"Было введено: ");
};

afx_msg void WINDOW::OnExit()
{
        int Answer;
        Answer = MessageBox("Quit the Program?","Exit", MB_YESNO
                                | MB_ICONQUESTION);
        if (Answer == IDYES)
                SendMessage(WM_CLOSE);
};

class APPLICATION : public CWinApp
{
public:
        BOOL InitInstance();
};

BOOL APPLICATION :: InitInstance()
{
        m_pMainWnd =  new WINDOW;
        m_pMainWnd -> ShowWindow(m_nCmdShow);
        m_pMainWnd -> UpdateWindow();
        return TRUE;
}

APPLICATION  TheApplication; .
```

Код приложения с пользовательским меню и системой диалоговых окон

```
#include <afxwin.h>
#include <iostream>
#include <string>
#include "resource.h"
using namespace std;

class aCDATA_DIALOG : public CDialog
{
public:
        aCDATA_DIALOG(char *DialogName, CWnd *Owner):
        CDialog(DialogName, Owner){};
        afx_msg void OnOK();
        afx_msg void OnCancel();
        DECLARE_MESSAGE_MAP()
};
```

```
BEGIN_MESSAGE_MAP(aCDATA_DIALOG, CDialog)
        ON_COMMAND(IDOK, OnOK)
        ON_COMMAND(IDCANCEL, OnCancel)
END_MESSAGE_MAP()

afx_msg void aCDATA_DIALOG::OnOK()
{
        MessageBox("OnOK","OnOK");
        CEdit *ebptr = (CEdit *) GetDlgItem(IDC_EDIT1);
        char Str[80];
        int I;
        I = ebptr->GetWindowText(Str, sizeof Str-1);
        MessageBox(Str,"Edit Box Contains");
        char Str1[80];
        int I1;
        I1 = GetDlgItemInt(IDC_EDIT1);
        wsprintf(Str1, "%d", I1);
        MessageBox(Str1,"Edit Box Contains");
        char Str2[80];
        int I2;
        float MyFloat;
        GetDlgItemText(IDC_EDIT1,Str2,strlen(Str2));
        MessageBox(Str2,"Edit Box Contains");
        MyFloat = atof(Str2);
};

afx_msg void aCDATA_DIALOG::OnCancel()
{
        MessageBox("OnCancel","OnCancel");
        int Answer;
        Answer = MessageBox("Уверены?","Cancel", MB_YESNO |
        if (Answer == IDYES)
                MB_ICONQUESTION);
        EndDialog(0);
};

class CSampleDialog : public CDialog
{
public:
        CSampleDialog(char *DialogName, CWnd *Owner):
        CDialog(DialogName, Owner){};
        afx_msg void OnFirstButton();
        afx_msg void OnSecondButton();
        DECLARE_MESSAGE_MAP()
```

```
};

BEGIN_MESSAGE_MAP(CSampleDialog, CDialog)
        ON_COMMAND(IDC_BUTTON1, OnFirstButton)
        ON_COMMAND(IDC_BUTTON2, OnSecondButton)
END_MESSAGE_MAP()

afx_msg void CSampleDialog::OnFirstButton()
{
        MessageBox("OnFirstButton","OnFirstButton");
};

afx_msg void CSampleDialog::OnSecondButton()
{
        MessageBox("OnSecondButton","OnSecondButton");
};

class CMainWin : public CFrameWnd
{
public:
        CMainWin();
        afx_msg void OnEdit();
        afx_msg void OnHelp();
        afx_msg void OnQuit();
        afx_msg void OnMessageBox();
        afx_msg void OnButtons();
        DECLARE_MESSAGE_MAP()
};

CMainWin::CMainWin()
{
        Create(NULL, "Окно с меню", WS_OVERLAPPEDWINDOW, rectDefault,
                NULL, (LPTSTR) IDR_MENU1);
}

afx_msg void CMainWin::OnEdit()
{
        MessageBox("OnEdit","OnEdit");
        aCDATA_DIALOG MyDataDialog((LPTSTR)IDD_DIALOG2 ,this);
        MyDataDialog.DoModal();
};

afx_msg void CMainWin::OnHelp()
{
        MessageBox("OnHelp","Help");
```

```
};

afx_msg void CMainWin::OnQuit()
{
	int Answer;
	Answer = MessageBox("Завершить?","Exit", MB_YESNO |
					MB_ICONQUESTION);
	if (Answer == IDYES)
		SendMessage(WM_CLOSE);
};

afx_msg void CMainWin::OnMessageBox()
{
	MessageBox("OnMessageBox","OnMessageBox");
};

afx_msg void CMainWin::OnButtons()
{
	CSampleDialog MyDialog((LPTSTR)IDD_DIALOG1 /*"SampleDialog"*/,this);
	MyDialog.DoModal();
};


BEGIN_MESSAGE_MAP(CMainWin, CFrameWnd)
	ON_COMMAND(ID_EXIT_QUIT, OnQuit)
	ON_COMMAND(ID_DIALOGBOXES_MESSAGEBOX, OnMessageBox)
	ON_COMMAND(ID_DIALOGBOXES_BUTTONS, OnButtons)
	ON_COMMAND(ID_DIALOGBOXES_EDIT, OnEdit)
	ON_COMMAND(ID_HELP, OnHelp)
END_MESSAGE_MAP()

class CApp : public CWinApp
{
public:
	BOOL InitInstance();
};

BOOL CApp::InitInstance()
{
	m_pMainWnd = new CMainWin;
	m_pMainWnd->ShowWindow(m_nCmdShow);
	m_pMainWnd->UpdateWindow();
	return TRUE;
}
CApp  App;    .
```

Лекция № 12 Разработка интерфейсов. Средства разработки приложений C#

1 Средства автоматизации

В Visual Studio файлы с исходным кодом и ресурсами (меню, панелями инструментов и диалоговыми окнами) группируются в проекты. Проект позволяет редактировать входящие в него файлы и управлять взаимосвязями между ними. Инструменты сборки проекта Visual C++ - компиляторы кода и ресурсов, компоновщик - настраиваются в диалоговом окне Project Settings. Для каждого проекта можно определить несколько независимых конфигураций их параметров. При использовании мастера AppWizard автоматически создаются конфигурации Debug и Release.

Проекты всегда находятся в рабочей области. По умолчанию одиночный проект создается в рабочей области, а обе его конфигурации называются одинаково. Для эффективной организации разработки полезно группировать связанные друг с другом проекты внутри одной рабочей области. Это позволяет устанавливать между ними взаимосвязи, обеспечивая согласованную сборку проектов с общими файлами.

Для автоматизации разработки приложений используются различные средства среды, часть из которых представлена ниже.

1. Мастера создания каркасов приложений. В том числе, на базе библиотеки MFC. Так в диалоговом окне New можно выбрать тип создаваемого проекта приложения, компонента или библиотеки. Мастера каркасов собраны на вкладке Projects. Это проекты (каркасы) MFC - программы на базе MFC, пригодные для разработки приложений, DLL-библиотек и элементов управления на базе ActiveX. Другие типы проектов, например, созданные мастером Internet Server API (ISAPI) Wizard, также используют каркас MFC-приложения. При использовании MFC необходимо либо компоновать статические библиотеки MFC с кодом, либо удостовериться в наличии DLL-библиотек MFC на всех компьютерах, где будет устанавливаться программа.

2. Средства графического отображения, просмотра компонентов приложений (классов, файлов, ресурсов проектов) с возможностью их редактирования. Для графического отображения компонентов - объектов рабочей области - используется список с древовидным отображением, корневыми узлами которого являются проекты. При этом содержимое проектов отображается тремя вкладками в окне рабочей области. Вкладка ClassView представляет программу в объектно-ориентированном виде, отображая классы, и их члены. Двойной щелчок класса или его члена вызовет переход к его объявлению или реализации. Вкладка ResourceView отображает ресурсы, сгруппированные по категориям. Двойной щелчок ресурса загружает соответствующий редактор. Вкладка FileView отображает все файлы проекта, которые можно редактировать. Двойной щелчок файла вызовет переход к его содержимому. Кроме этого, все вкладки поддерживают контекстное меню, вызываемое левой клавишей мыши.

3. Редакторы ресурсов для создания (описания) и редактирования графических ресурсов (меню, панелей инструментов, пиктограмм и т.п.).

4. Мастер создания и редактирования классов и обработчиков сообщений ClassWizard (Ctrl-W).

5. Средство Complete Word, обеспечивающее автоматический донабор ключевых слов, имен, а также контекстную подсказку по классам, методам, атрибутам. Это позволяет в процессе создания-редактирования исходных кодов оперативно получать подсказку в виде прототипов методов, выбирать из предложенных списков необходимые члены классов (методы, атрибуты). Выбор осуществляется либо просмотром списков либо набором первых символов из их названий. По этой же причине атрибуты класса, особенно пользовательские, начинаются префиксом m_, что обеспечивает их группировку в описаниях классов.

Обновление меню. В процессе работы с приложением, которое использует меню, элементы управления (ЭУ), может меняться как состояние самого меню (пунктов меню) так и отдельных ЭУ. Например, пункты меню могут помечаться при выборе, могут блокироваться и т.д.

Например, необходимо управлять блокировкой пунктов меню управления активизацией немодального окна - DialogsNotmodal, DialogsNotmodaldel. Для этого:

1. Добавьте в автоматический каркас mfc-приложения в класс class CChildView : public CWnd переменную состояния (флаг) немодального окна public: int m_NotModalState.

2. В конструкторе CChildView( ) установите начальное значение m_NotModalState = 0;

3. В обработчике OnDialogsNotmodal( ) задавайте значение m_NotModalState = 1.

4. В обработчике OnDialogsNotmodaldel( ) задавайте значение m_NotModalState = 0.

5. Добавьте обработчики сообщений обновления меню (рисунок 62) (сообщение UPDATE_COMMAND_UI) для обоих пунктов меню (DialogsNotmodal и DialogsNotmodaldel).

Прототипы обработчиков void CChildView::OnUpdateDialogsNotmodal(CCmdUI* pCmdUI) и void CChildView::OnUpdateDialogsNotmodaldel(CCmdUI* pCmdUI) .

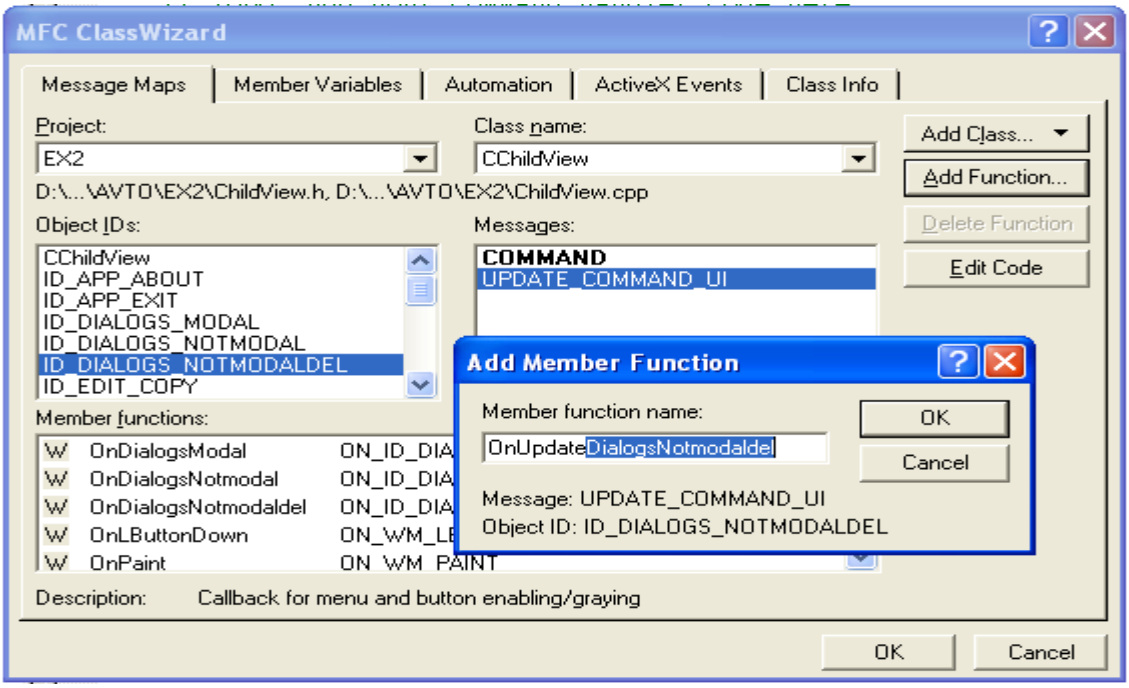


Рисунок 62 - Добавление обработчика

6. Добавьте коды в обработчики в соответствии с примерами ниже

void CChildView::OnUpdateDialogsNotmodal(CCmdUI* pCmdUI)

```
{
    // TODO: Add your command update UI handler code here
    if (m_NotModalState == 1)
    pCmdUI->Enable(FALSE);
}
void CChildView::OnUpdateDialogsNotmodaldel(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    if (m_NotModalState == 0)
    pCmdUI->Enable(FALSE);
}  .
```

Тогда при запуске приложения меню высвечивается изначально с блокировкой пункта NotModalDel. Однако после создания-активизации модального окна пунктом NotModal пункт меню NotModalDel разблокируется, а пункт NotModal наоборот – станет не активным (рисунки 63, 64).

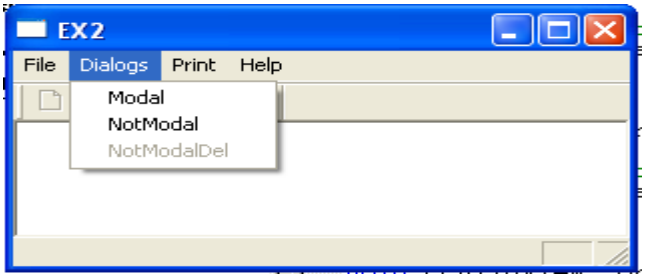


Рисунок 63 - Пункты динамического меню

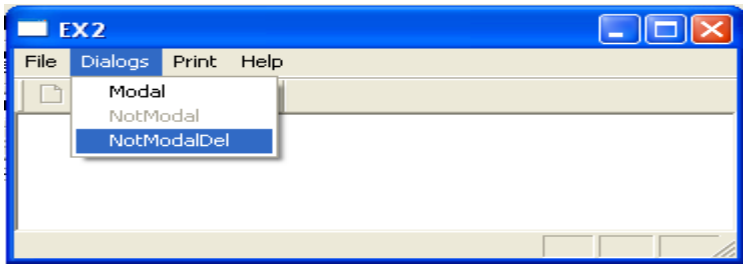


Рисунок 64 - Пункты динамического меню

Использование переменных для управления ЭУ. Каждому элементу управления можно поставить в соответствие переменные, которые будут использоваться: - для обмена данными с ЭУ; - для управления ЭУ (членами класса, обеспечивающего функциональность ЭУ). Для этого используются переменные из категорий Value или Control.

Для обмена данными используются переменные встроенных типов C++, например, CString m_E1 (рисунок 65). Между такой переменной, если она назначена - связана с ЭУ, и самим ЭУ система устанавливает “информационный канал связи” с автоматической передачей данных из ЭУ в переменную методом UpdateData(TRUE) и обратно методом UpdateData(FALSE). Этот метод с параметром TRUE вызывается также и автоматиче-

ски, например, при инициализации диалогового окна, при нажатии кнопки с IDOK при использовании ее стандартного обработчика.

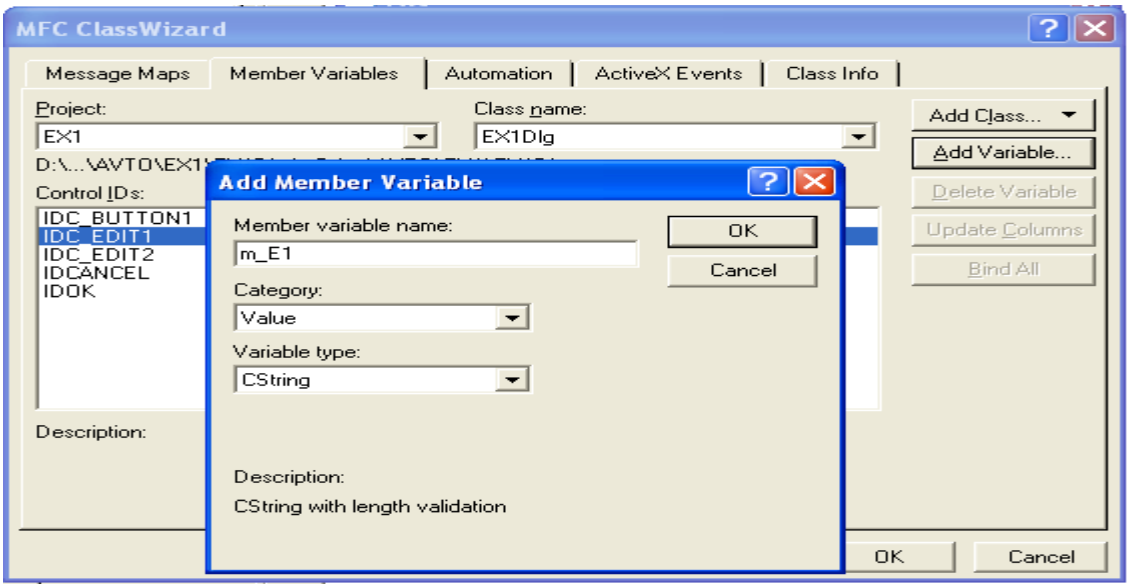


Рисунок 65 - Добавление переменной для ЭУ

Для управления используются переменные-объекты соответствующего класса ЭУ, например, CEdit m_cE1. Такая переменная, если она назначена - связана с ЭУ, применяется для непосредственного вызова методов его класса. Автоматизация связывания переменных и ЭУ реализуется мастером ClassWizard. Мастер вызывается из контекстных меню либо сочетанием клавиш Ctrl-W. При этом в коде программы в методе ДО DoDataExchange будут автоматически сгенерированы выделенные ниже строки с вызовом функций, обеспечивающих установление каналов связи

```
void EX1Dlg::DoDataExchange(CDataExchange* pDX)
{
        CDialog::DoDataExchange(pDX);
        //{{AFX_DATA_MAP(EX1Dlg)
        DDX_Control(pDX, IDC_EDIT2, m_cE2);
        DDX_Control(pDX, IDC_EDIT1, m_cE1);
        DDX_Text(pDX, IDC_EDIT1, m_E1);
        DDX_Text(pDX, IDC_EDIT2, m_E2);
        //}}AFX_DATA_MAP
}  .
```

Такие строки выделяются в помеченный комментариями сегмент

```
//{{AFX_DATA_MAP(EX1Dlg)     …
//}}AFX_DATA_MAP  .
```

Переменные, связанные с ЭУ можно применять для их инициализации. Например, используйте переменные для инициализации окон редактирования группы ДО - Edits. Для этого в метод BOOL EX1Dlg::OnInitDialog( ) внесите изменения. Например, используя связанные переменные

```
// TODO: Add extra initialization here
        m_E2 = "echo";
        m_E1 = "enter";
        UpdateData(FALSE);
```

или связанные с ЭУ управляющие переменные

```
// TODO: Add extra initialization here
        m_cE2.SetWindowText("echo");
        m_cE1.SetWindowText("enter");   .
```

2 Авто-каркасы

Библиотека MFC поддерживает разнообразные каркасы, используемые при создании пользовательских приложений. Каркасы могут быть построены вручную либо автоматически, если в системе программирования есть специальные программы – мастера, генерирующие тексты каркасов в соответствии с заданными пользователем требованиями, параметрами. Полученные каркасы затем дорабатываются с целью придания проекту необходимой функциональности и реализации разрабатываемого приложения. В системе программирования Visual Studio C++ ряд типовых, широко используемых каркасов можно получить, в частности, с помощью мастера MFC AppWizard (exe). Такие каркасы и их характеристики представлены рисунками 66, 67.

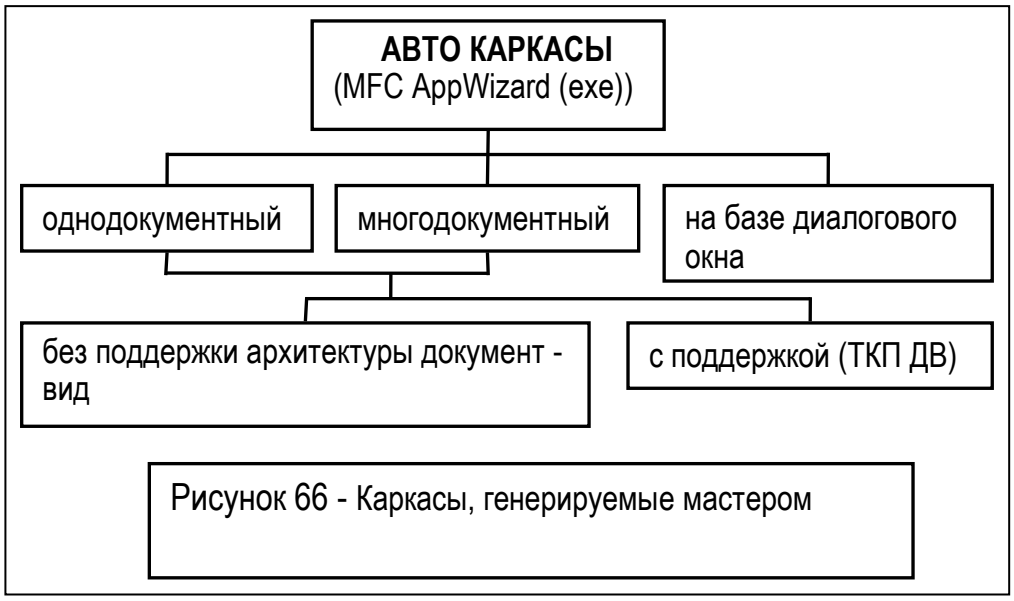


Рисунок 66 - Каркасы, генерируемые мастером

Это одно и многодокументные каркасы соответственно с интерфейсом SDI (Single Document Interface) и MDI (Multiple Document Interface) и каркасы с интерфейсом на базе

диалогового окна, используемого в качестве главного. Первые два каркаса могут базироваться на одной из двух архитектур приложения. Это типовой каркас приложения (ТКП) - архитектура создания приложений - “окно с рамкой” (CFrameWnd) в роли главного для организации всех функций по обработке данных (документов), включая их хранение, визуализацию. Либо типовой каркас приложения (ТКП ДВ) - архитектура создания приложений документ-вид, рассматриваемый ниже.

Каркас документ-вид. Здесь объект обработки разделяется на объекты класса ВИД (производного от класса CView) и объекты класса ДОКУМЕНТ (производного от класса CDocument). С каждым из документов может быть связан один или несколько объектов классов, производных от класса ВИД.

При этом функциональность класса ВИД обеспечивает, определяет оконный вид, облик, представление документа (на экране, при печати), а функциональность класса ДОКУМЕНТ обеспечивает типовые действия по работе с данными документа и, таким образом, позволяет представлять более абстрактные объекты, чем документ, понимаемый как объект обработки текстового процессора.

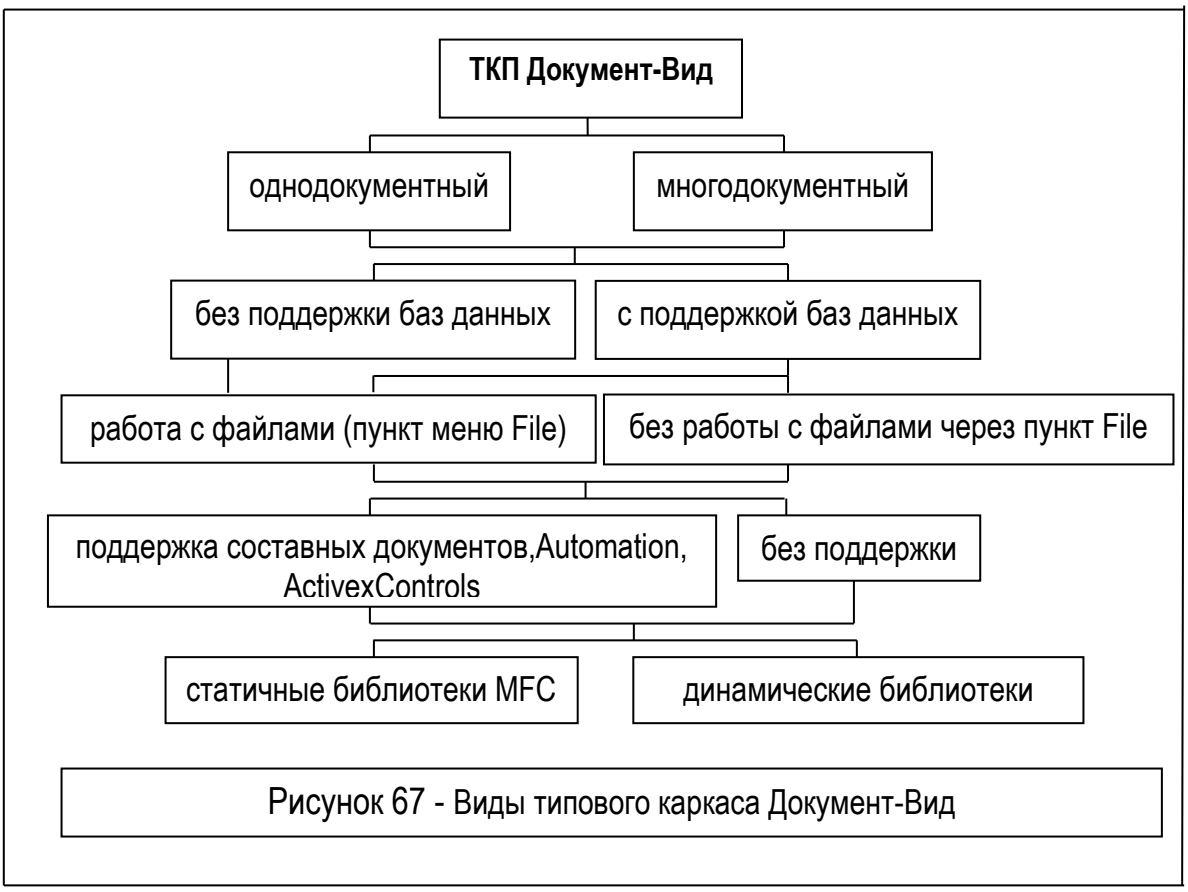


Рисунок 67 - Виды типового каркаса Документ-Вид

Базовая специализация указанной архитектуры (ТКП ДВ) – разработка приложений для создания, редактирования, хранения, печати и просмотра документа (-ов), в том числе в различных форматах отображения. Более развернуто разновидности авто каркасов ТКП ДВ, получаемые при различных настройках, задаваемых пользователем при генерации, представлены на рисунке 67. Отличительной особенностью всех этих каркасов является поддержка механизма сериализации документов – типовых пунктов подменю File для автоматической загрузки документа для работы из выбранного файла и автоматической выгрузки в выбранный файл по завершении работы. Каркасы ТКП ДВ могут использовать технологии работы с составными документами, позволяя работать с разнотипными документами.

Каркас включает: 1) программу на языке C++, описывающую функциональность каркаса в виде набора классов. Описания, декомпозированные на отдельные файлы, образуют программную составляющую проекта; 2) “графические” ресурсы, образующие основу пользовательского интерфейса. Их описания собраны в специальном файле и хранятся отдельно от программной составляющей проекта. При этом главное меню, пиктограмма и панель инструментов приложения имеют общий идентификатор (здесь IDD_MAINFRAME); 3) документы, для обработки которых создается приложение (каркас). Документы хранятся во внешней памяти и, при необходимости, загружаются в память с использованием механизма сериализации.

Описания, декомпозированные на отдельные файлы, образуют программную составляющую проекта. Она представлена на рисунах 68, 69 в виде диаграмм программных компонентов. Диаграмма классов каркаса документ-вид представлена на рисунке 70. Здесь жирным шрифтом выделены пользовательские классы и их базовые классы из библиотеки MFC. Взаимодействие объектов каркаса при запуске и инициализации приложения представлены диаграммой последовательностей на рисунке 71.

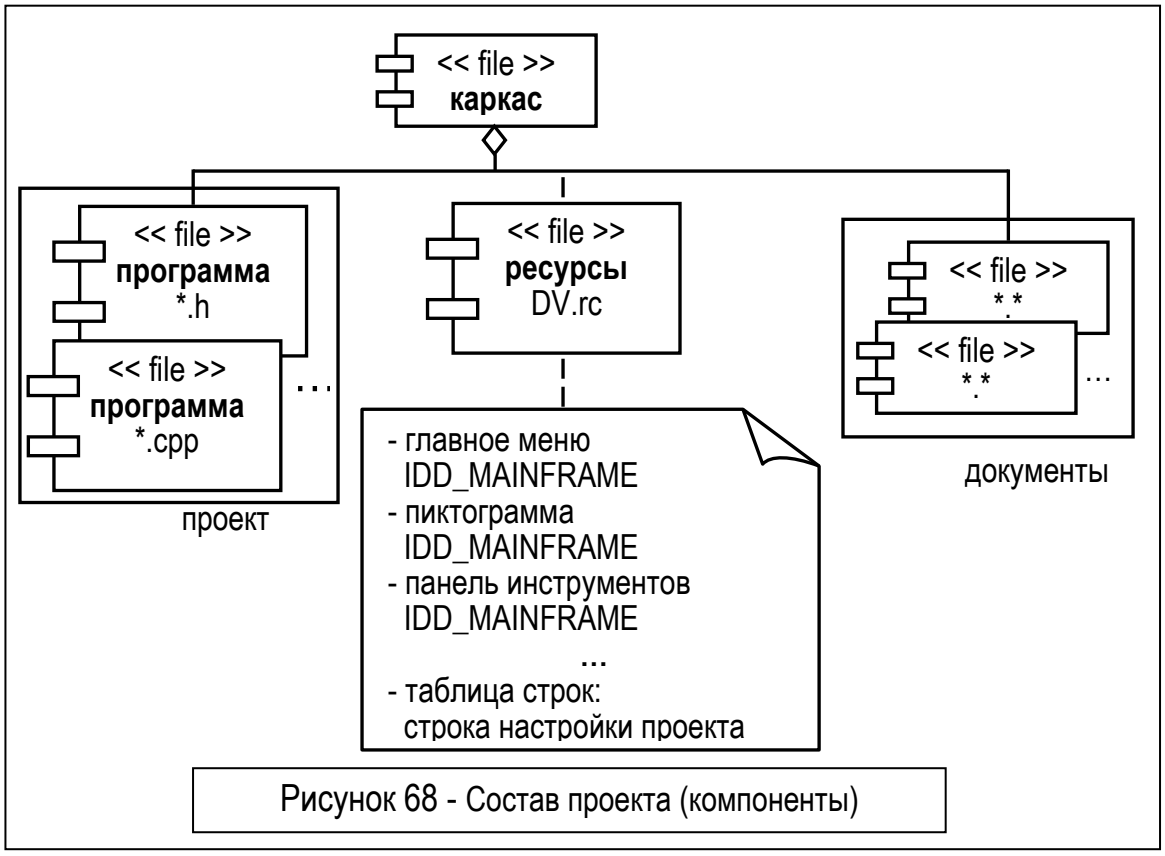


Рисунок 68 - Состав проекта (компоненты)

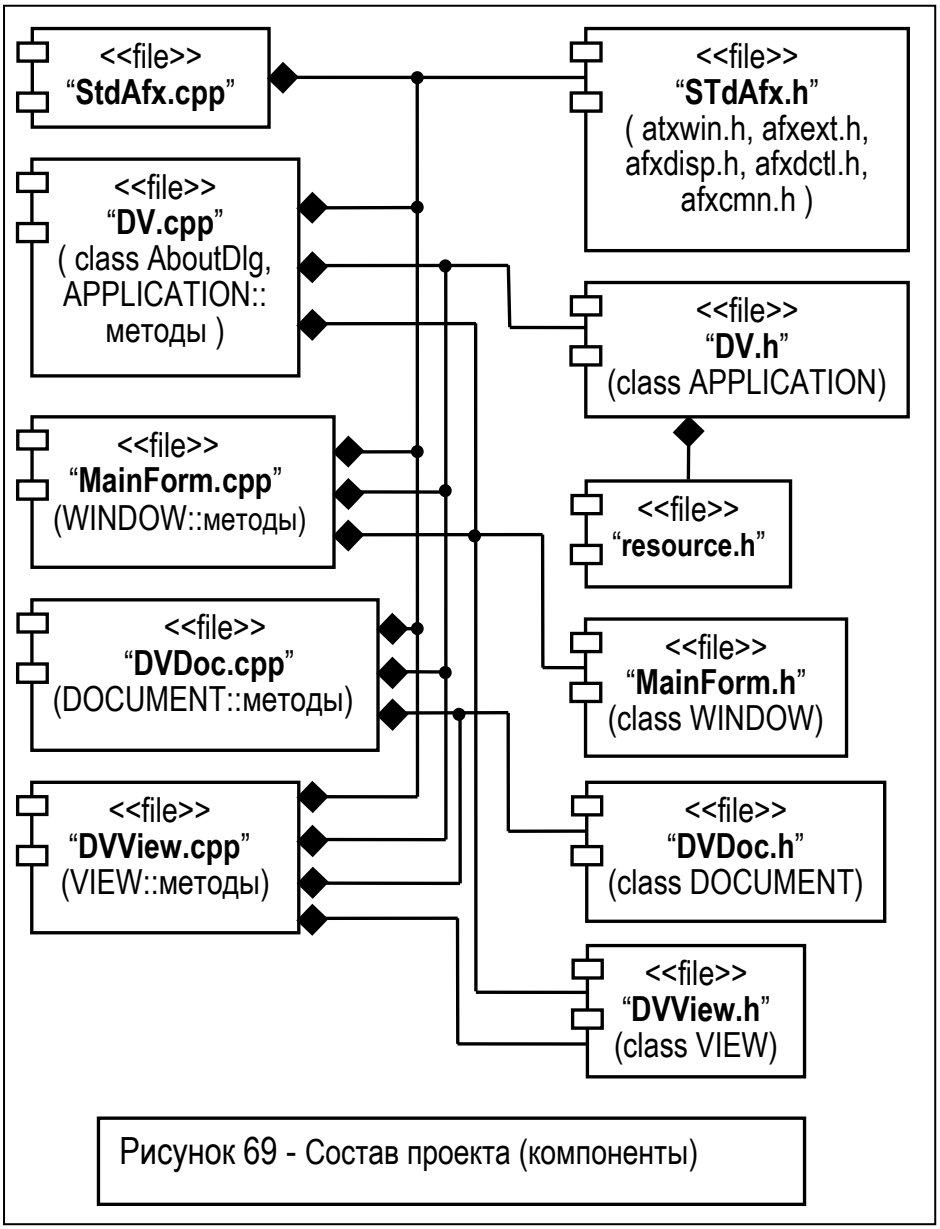


Рисунок 69 - Состав проекта (компоненты)

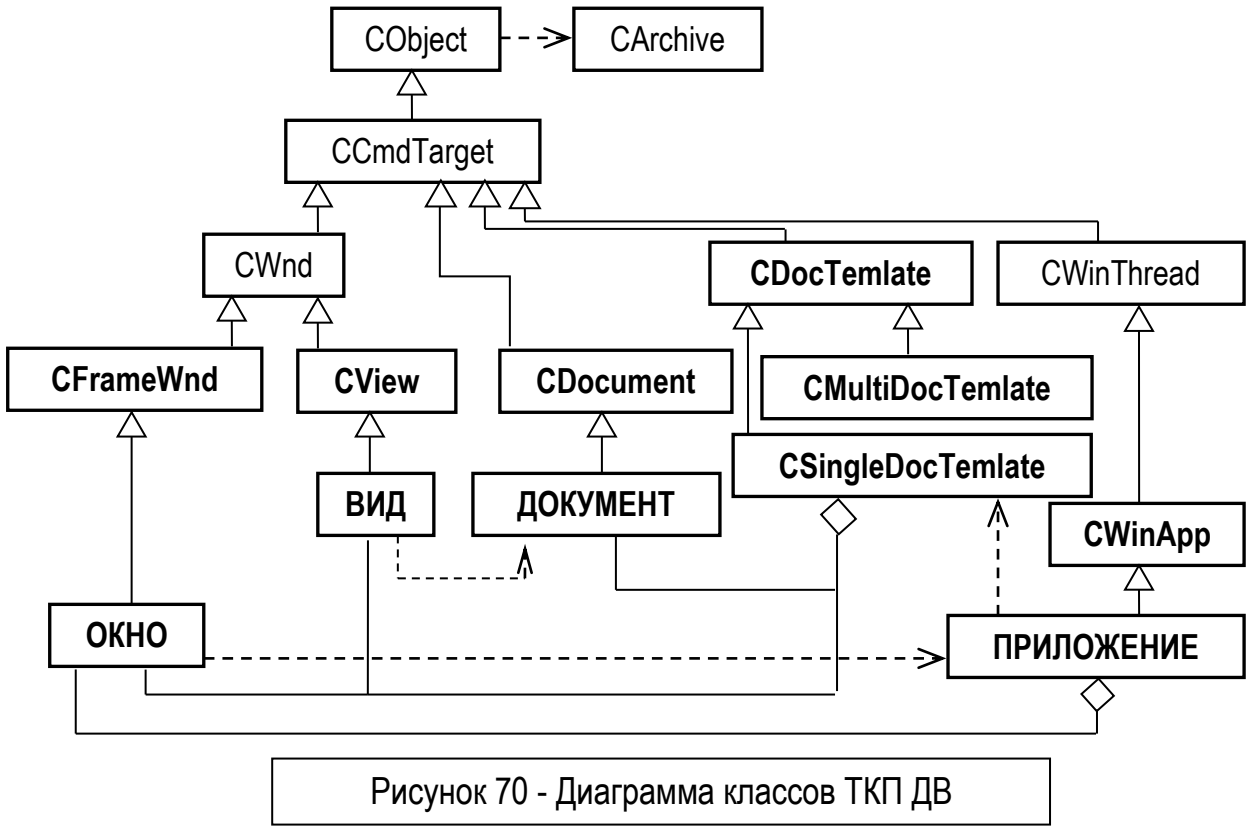


Рисунок 70 - Диаграмма классов ТКП ДВ

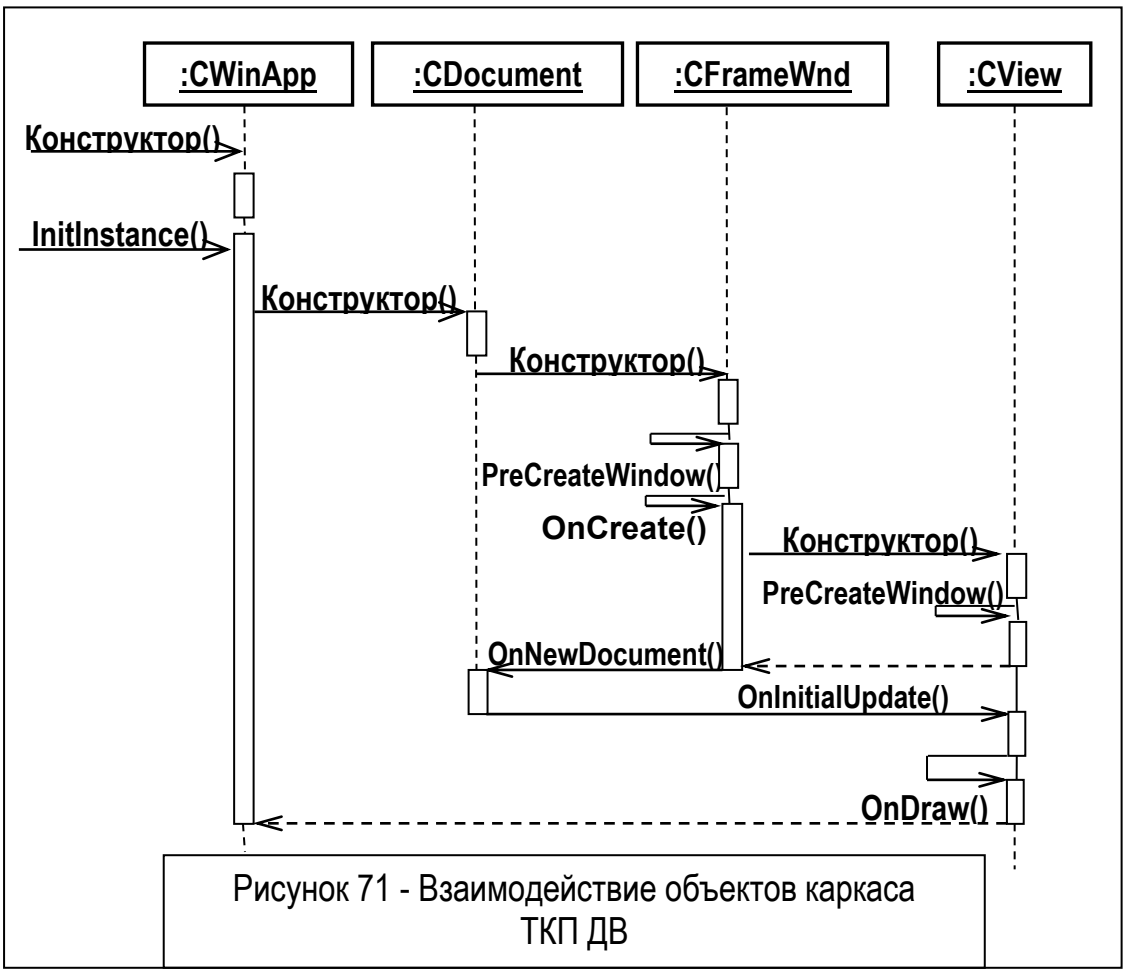


Рисунок 71 - Взаимодействие объектов каркаса ТКП ДВ

3 Платформа MS.NET

Структура платформы для разработки .NET приложений приведена ниже (рисунок 72). Базируется на операционной системе Windows. Обеспечивает пользователю рабочую среду (.NET Framework) – то есть как средства разработки приложений так и виртуальную машину для их выполнения.

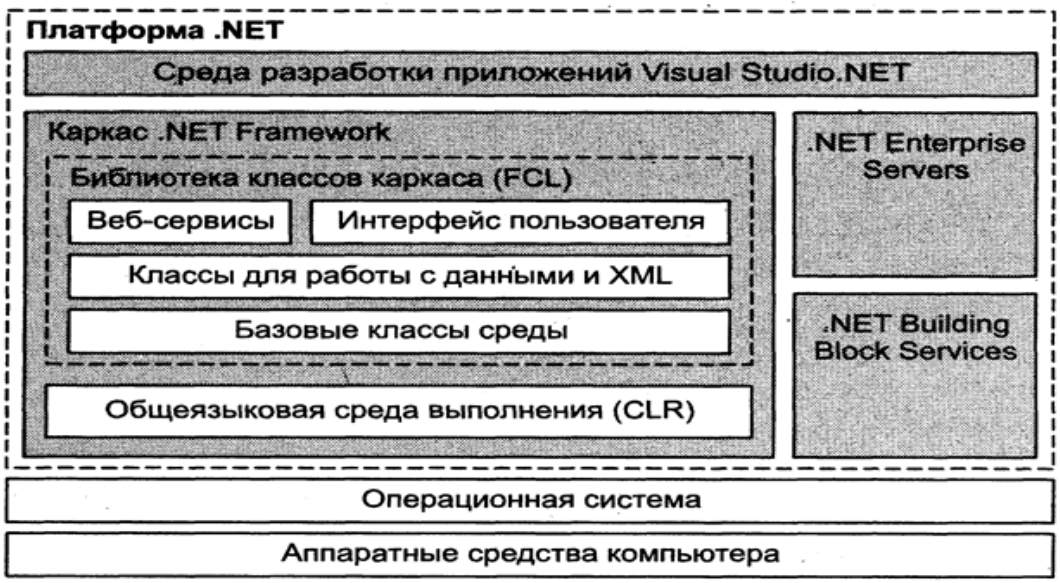


Рисунок 72 - Структура платформы MS.NET

Язык C++ можно рассматривать как развитие промежуточного языка visual C++ / CLI, также ориентированного на повышении продуктивности разработки приложений и обеспечивающего безопасность кода и данных. Ключевые слова, используемые в языке visual C#, приведены в таблице 30.

Таблица 30 - Ключевые слова C#

abstract	as	base	bool	break
byte	case	catch	char	checked
class	const	continue	decimal	default
delegate	do	double	else	enum
event	explicit	extern	false	finally
fixed	float	for	foreach	goto
if	implicit	in	int	interface
internal	is	lock	long	namespace
new	null	object	operator	out
override	params	private	protected	public
readonly	ref	return	sbyte	sealed
short	sizeof	stackalloc	static	string
struct	switch	this	throw	true
try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	void
volatile	while			

Схема выполнения .NET-приложений показана ниже (рисунок 73).

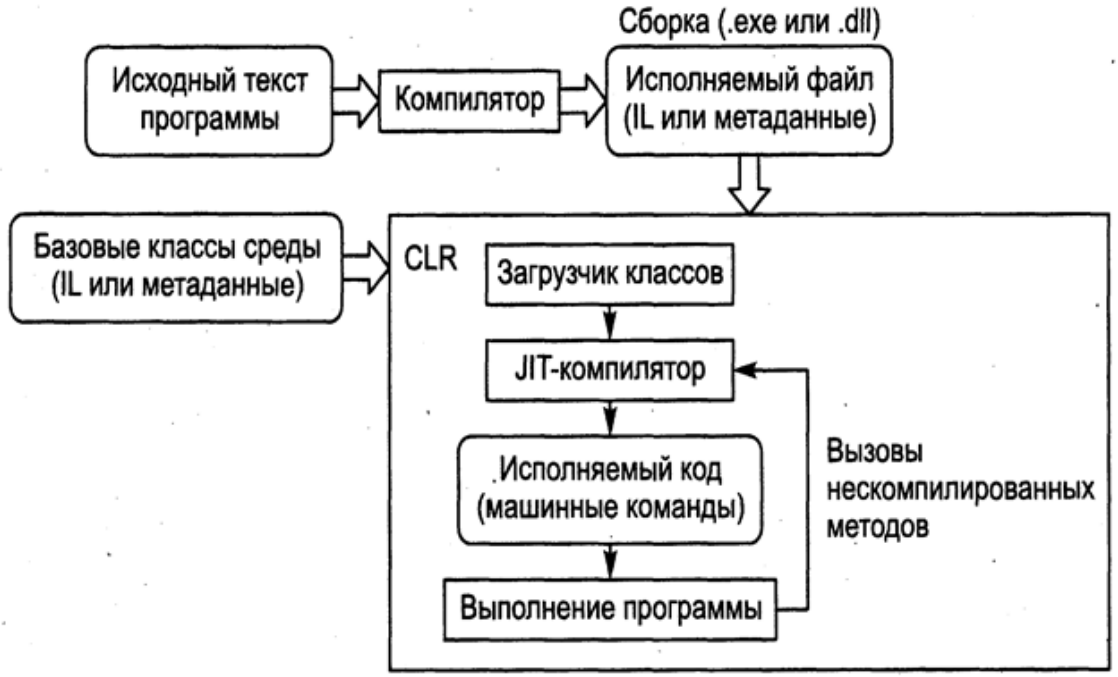


Рисунок 73 - Схема выполнения программы в .NET

Указанные средства (платформа и язык) обеспечивают создание безопасного кода и данных. Одной из наиболее важных функциональных возможностей для обеспечения безопасного кода является средство сборки мусора (garbage collection). Механизм сборки мусора в .NET функционирует следующим образом: средство сборки мусора инспектирует память компьютера время от времени и удаляет из нее все, что уже не используется к текущему моменту времени. Никаких установленных временных рамок для выполнения этой операции нет. Это гарантирует уверенность в полном очищении использованной части памяти после завершения ее эксплуатации без вмешательства пользователя.

4 Каркас консольного приложения

Шаблон (каркас) для создания консольных приложений представлен кодом ниже. Он базируется на использовании класса (здесь это Class1 – имя может меняться пользователем), который содержит статичный метод (здесь Main), помеченный атрибутом [STAThread] для операционной системы как точка входа при запуске приложения

```
using System;

namespace ConsoleApplication1
{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class Class1
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main(string[] args)
        {
            //
            // TODO: Add code to start application her
            //
        }
    }
}
```

Лекция № 13 Типы данных .NET

1 Характеристика типов данных

Классификация типов данных представлена ниже (рисунки 74, 75).

Рисунок 74 - Классификация типов данных C#

Рисунок 75 - Классификация типов данных C# по способу хранения переменных

В языке используются:
- типы-значения:
1 – скалярные, базовые, встроенные типы;
2 – структуры, перечисления;
- ссылочные (дескрипторные) типы:
3 – классы,
4 - массивы,
5 - строки,
6 - делегаты,
7 – интерфейсы;
- указатели (pointer).

Встроенные типы C# и их соответствие системе типов CTS представлены в таблице ниже (таблица 31).

Данные ссылочных типов размещаются в “управляемой” куче.

2 Скалярные типы, преобразование типов

Для преобразования типов данных можно использовать методы различных классов системной библиотеки.

Так преобразование можно выполнить либо с помощью специального класса Convert, определенного в пространстве имен System, либо с помощью метода Parse.

Примеры ввода и преобразования данных скалярных типов C# приведены ниже.

Таблица 31 - Встроенные типы C#

Логический тип	bool	Boolean	true, false		
Целые типы	sbyte	SByte	От –128 до 127	Со знаком	8
	byte	Byte	От 0 до 255	Без знака	8
	short	Int16	От –32 768 до 32 767	Со знаком	16
	ushort	UInt16	От 0 до 65 535	Без знака	16
	int	Int32	От $-2 \cdot 10^9$ до $2 \cdot 10^9$	Со знаком	32
	uint	UInt32	От 0 до $4 \cdot 10^9$	Без знака	32
	long	Int64	От -9×10^{18} до $9 \cdot 10^{18}$	Со знаком	64
	ulong	UInt64	От 0 до $18 \cdot 10^{18}$	Без знака	64
Символьный тип	char	Char	От U+0000 до U+ffff	Unicode-символ	16
Вещественные[1]	float	Single	От $1.5 \cdot 10^{-45}$ до $3.4 \cdot 10^{38}$	7 цифр	32
	double	Double	От $5.0 \cdot 10^{-324}$ до $1.7 \cdot 10^{308}$	15–16 цифр	64
Финансовый тип	decimal	Decimal	От $1.0 \cdot 10^{-28}$ до $7.9 \cdot 10^{28}$	28–29 цифр	128
Строковый тип	string	String	Длина ограничена объемом доступной памяти	Строка из Unicode-символов	
Тип object	object	Object	Можно хранить все что угодно	Всеобщий предок	

```
using System;
namespace ConsoleApplication1
{    class Class1
    {    static void Main()
        {
            Console.WriteLine( "Введите строку" );
            string s = Console.ReadLine();                  // 1
            Console.WriteLine( "s = " + s );

            Console.WriteLine( "Введите символ" );
            char c = (char)Console.Read();                  // 2
            Console.ReadLine();                             // 3
            Console.WriteLine( "c = " + c );
            string buf;        // строка - буфер для ввода чисел
            Console.WriteLine( "Введите целое число" );
            buf = Console.ReadLine();
            int i = Convert.ToInt32( buf );                 // 4
            Console.WriteLine( i );

            Console.WriteLine( "Введите вещественное число" );
            buf = Console.ReadLine();
            double x = Convert.ToDouble( buf );             // 5
            Console.WriteLine( x );

            Console.WriteLine( "Введите вещественное число" );
            buf = Console.ReadLine();
            double y = double.Parse( buf );                 // 6
            Console.WriteLine( y );

            Console.WriteLine( "Введите вещественное число" );
            buf = Console.ReadLine();
            decimal z = decimal.Parse( buf );               // 7
            Console.WriteLine( z );
        }
    }
}
```

3 Организация вычислений

В языке C# используется привычный набор команд для организации ветвлений и циклов. Добавлена команда foreach (...). Порядок использования команды (в синтаксисе CLI) описан ниже

```
for each (wchar_t ch in proverb)
{   // обработка текущего символа строки, сохраненного в переменной ch   .}
```

Здесь символы в строке proverb представлены в кодировке Unicode, поэтому для хранения каждого из них применяется переменная типа wchar_t (эквивалент типа Char). При первой итерации переменная ch содержит первый символ строки, при второй итерации следующий символ и т.д., пока не будут выделены и обработаны все символы строки.

Для организации вычислений можно использовать методы из пространства имен System.Math (таблица 32).

Таблица 32 - Члены класса Math

Имя	Описание	Результат	Пояснения
Abs	Модуль	Перегружен[1]	$\|x\|$ записывается как Abs(x)
Acos	Арккосинус[2]	double	Acos(double x)
Asin	Арксинус	double	Asin(double x)
Atan	Арктангенс	double	Atan(double x)
Atan2	Арктангенс	double	Atan2(double x, double y) — угол, тангенс которого есть результат деления y на x
BigMul	Произведение	long	BigMul(int x, int y)
Ceiling	Округление до большего целого	double	Ceiling(double x)
Cos	Косинус	double	Cos(double x)
Cosh	Гиперболический косинус	double	Cosh(double x)
DivRem	Деление и остаток	Перегружен	DivRem(x, y, rem)
E	База натурального логарифма (число e)	double	2,71828182845905
Exp	Экспонента	double	e^x записывается как Exp(x)
Floor	Округление до меньшего целого	double	Floor(double x)
IEEERemainder	Остаток от деления	double	IEEERemainder(double x, double y)
Log	Натуральный логарифм	double	$\log_e x$ записывается как Log(x)

Log10	Десятичный логарифм	double	$\log_{10} x$ записывается как Log10(x)
Max	Максимум из двух чисел	Перегружен	Max(x, y)
Min	Минимум из двух чисел	Перегружен	Min(x, y)
PI	Значение числа π	double	3,14159265358979
Pow	Возведение в степень	double	x^y записывается как Pow(x, y)
Round	Округление	Перегружен	Round(3.1) даст в результате 3, Round (3.8) даст в результате 4
Sign	Знак числа	int	Аргументы перегружены
Sin	Синус	double	Sin(double x)
Sinh	Гиперболический синус	double	Sinh(double x)
Sqrt	Квадратный корень	double	$\sqrt{x}$ записывается как Sqrt(x)
Tan	Тангенс	double	Tan(double x)
Tanh	Гиперболический тангенс	double	Tanh(double x)

При выполнении операций в C# генерируются исключения. Так, например, если необходимый для хранения объекта объем памяти выделить не удалось, то генерируется исключение System.OutOfMemoryException; если оба операнда целочисленные или типа decimal и результат операции слишком велик для представления с помощью заданного типа, генерируется исключение System.OverflowException; если оба операнда целочисленные, результат операции округляется до ближайшего целого числа – при этом если делитель равен нулю, то генерируется исключение System.DivideByZeroException и т.д. Таблица типовых исключений (таблица 33) и формат их использования приведены ниже.

Таблица 33. Стандартные исключения

Имя	Описание
ArithmeticException	Ошибка в арифметических операциях или преобразованиях (является предком DivideBeZeroException и OverFlowException)
ArrayTypeMismatchException	Попытка сохранения в массиве элемента несовместимого типа
DivideByZeroException	Попытка деления на ноль
FormatException	Попытка передать в метод аргумент неверного формата
IndexOutOfRangeException	Индекс массива выходит за границы диапазона
InvalidCastException	Ошибка преобразования типа
OutOfMemoryException	Недостаточно памяти для создания нового объекта
OverFlowException	Переполнение при выполнении арифметических операций
StackOverFlowException	Переполнение стека

```
try {
   ...  // Контролируемый блок
}
catch ( OverflowException e ) {
    ...    // Обработка исключений класса OverflowException (переполнение)
}
catch ( DivideByZeroException ) {
    ... // Обработка исключений класса DivideByZeroException (деление на 0)
}
catch {
    ...    // Обработка всех остальных исключений
}
```

Код примера использования исключений для проверки результатов ввода представлен ниже:

```
using System;
namespace ConsoleApplication1
{  class Program
    {   static void Main()
        {
            string buf;
            double u, i, r;
            try
            {
                Console.WriteLine( "Введите напряжение:" );
                u = double.Parse( Console.ReadLine() );

                Console.WriteLine( "Введите сопротивление:" );
                r = double.Parse( Console.ReadLine() );

                i = u / r;
                Console.WriteLine( "Сила тока - " + i );
            }
            catch ( FormatException )
            {
                Console.WriteLine( "Неверный формат ввода!" );
            }
            catch                                              // общий случай
            {
                Console.WriteLine( "Неопознанное исключение" );
            }
        }
    }
}
```

4 Консольный ввод-вывод

Для вывода данных используется статичный метод WriteLine(…). Ниже представлены примеры использования методов (в синтаксисе CLI).

```
Console::WriteLine(L"Имеется {0} пакетов весом {1:F2} фунтов.",
                   packageCount, packageWeight);
```

В подстроке {1:F2} двоеточие отделяет значение индекса (1), идентифицирующее аргумент, от следующего за ним указания формата (F2). Буква F в определении формата указывает, что вывод должен быть в форме ±ddd.dd... (где d представляет десятичную цифру, а 2 — это количество разрядов после точки). Вывод данных будет таким: Имеется 25 пакетов весом 7.50 фунта.

Общий формат вывода задается как { n, w : Axx } , где n – номер аргумента вывода в списке вывода (0, 1, 2 …), w – необязательный параметр – ширина поля вывода, A - формат вывода, берется из множества {C, D, E, F, X …}, xx - необязательный параметр – точность вывода, это часть поля вывода, отводимая под дробную часть данного.

Для ввода данных используются методы ReadLine, Read, ReadKey.

Пример использования метода Console::ReadLine() - String^ line = Console::ReadLine( ).

Результат - читается входная строка текста, завершающаяся нажатием клавиши Enter. Переменная line хранит ссылку на введенную строку.

Метод, читающий один символ с клавиатуры, выглядит как char ch = Console::Read( ).

С помощью этой функции можете читать входные данные символ за символом, а затем проанализировать прочитанные символы и преобразовать их в соответствующие числовые значения.

Метод Console::ReadKey() возвращает код нажатой клавиши в виде объекта класса ConsoleKeyInfo, определенного в пространстве имен System, например ConsoleKeyInfo keyPress = Console::ReadKey(true).

При вводе числового данного в виде строки, например, как String Str = Console.ReadLine( ), необходимо выполнить преобразование типа, например, как int Value = Int32.Parse(Str).

Лекция № 14 Значимые типы

1 Работа с данными символьного типа

Тип данных Char предназначен для хранения символов в кодировке Unicode. Ниже представлены базовые методы соответствующего класса System.Char (таблица 34).

Таблица 34 - Базовые методы работы с символами

Метод	Описание
GetNumericValue	Возвращает числовое значение символа, если он является цифрой, и –1 в противном случае
GetUnicodeCategory	Возвращает категорию Unicode-символа[1]
IsControl	Возвращает true, если символ является управляющим
IsDigit	Возвращает true, если символ является десятичной цифрой
IsLetter	Возвращает true, если символ является буквой
IsLetterOrDigit	Возвращает true, если символ является буквой или цифрой
IsLower	Возвращает true, если символ задан в нижнем регистре
IsNumber	Возвращает true, если символ является числом (десятичным или шестнадцатеричным)
IsPunctuation	Возвращает true, если символ является знаком препинания
IsSeparator	Возвращает true, если символ является разделителем
IsUpper	Возвращает true, если символ записан в верхнем регистре
IsWhiteSpace	Возвращает true, если символ является пробельным (пробел, перевод строки и возврат каретки)
Parse	Преобразует строку в символ (строка должна состоять из одного символа)
ToLower	Преобразует символ в нижний регистр
ToUpper	Преобразует символ в верхний регистр
MaxValue, MinValue	Возвращают символы с максимальным и минимальным кодами (эти символы не имеют видимого представления)

Примеры работы с данными символьного типа приведены ниже в синтаксисе CLI:

```
int main (array<System::String> ^ args)
{
    wchar_t  Letter;
    Console::Write(L"Введите букву: ");
    Letter = Console::Read ();
    if (Letter >= 'A')   if (Letter <= 'Z')
      {
          Console::WriteLine(L"Прописная буква "); return 0;
      }
   if (Letter >= 'a')  if (Letter <= 'z')
     {
        Console::WriteLine(L"Строчная буква "); return 0;
     }
  Console::WriteLine(L"Введена не буква ");   return 0;
}
```

```
int main (array<System::String> ^ args)
{
    Console::WriteLine(L"Нажмите комбинацию клавиш – а для выхода Esc ");
    ConsoleKeyInfo keyPress;
    do
    {
        keyPress = Console::ReadKey (true);
        Console::Write(L"Вы нажали ");
        if (safe_cast<int>(keyPress.Modifiers)>0)
         Console::Write(L" {0}, " , keyPress.Modifiers);
         Console::Write(L"{0}, что означает символ {1} ", keyPress.Key,
                    keyPress.KeyChar);
    } while ( keypress.Key != ConsoleKey::Escape);
return 0;
}

int main (array<System::String> ^ args)
{
    int vow(0), cons(0);
    String^ proverb (L" A nod is as good as a wink to a blind horse.");
    for each (wchar_t ch in proverb )
    {
        if ( Char::IsLetter (ch))
        {
            ch = Char::ToLower (ch);
            switch (ch)
            {
                case 'a': case 'e': case 'i': case 'o': case 'u': ++ vow; break;
                default: ++cons; break;
             }
        }
    }
    Console::WriteLine(proverb);
    Console::WriteLine(L" Строка proverb включает {0} гласных и {1} согласных букв",
                        vow, cons);
    return 0;
); .
```

2 Массивы

Массивы в языке C# организованы на основе базового класса Array. Они относятся к ссылочным типам данных, то есть располагаются в динамической области памяти, поэтому создание массива начинается с выделения памяти под его элементы. Элементами массива могут быть значения как значимых, так и ссылочных типов (в том числе массивов). Массив значимых типов хранит значения, а массив ссылочных типов — ссылки на элементы данных. Всем элементам при создании массива присваиваются значения по умолчанию: нули для значимых типов и ссылка null — для ссылочных. Применяются одномерные (рисунок 76), прямоугольные (рисунок 77) и ступенчатые (не выровненные) массивы (рисунок 78).

Варианты описания одномерного массива приведены ниже:

тип[ ] имя;
тип[ ] имя = { списки_инициализаторов };
тип[ ] имя = new тип[размерность];
тип[ ] имя = new тип[ ] { списки_инициализаторов };
тип[ ] имя = new тип[размерность] { списки_инициализаторов }; .

```
int[] a;                                  // 1   элементов нет
int[] b = new int[4];                     // 2   элементы равны 0
int[] c = { 61, 2, 5, -9 };               // 3   new подразумевается
int[] d = new int[] { 61, 2, 5, -9 };     // 4   размерность вычисляется
int[] e = new int[4] { 61, 2, 5, -9 };    // 5   избыточное описание
```

Рисунок 76 - Одномерные массивы

Прямоугольный массив имеет более одного измерения. Варианты описания двумерного массива приведены ниже:

тип[,] имя = new тип [размер_1, размер_2];
тип[,] имя = { списки_инициализаторов };
тип[,] имя = new тип[,] { списки_инициализаторов };
тип[,] имя = new тип [размер_1, размер_2] { списки_инициализаторов }; .

```
int[,] a;                                       // 1  элементов нет
int[,] b = new int[2, 3];                       // 2  элементы равны 0
int[,] c = {{1, 2, 3}, {4, 5, 6}};              // 3  new подразумевается
int[,] c = new int[,] {{1, 2, 3}, {4, 5, 6}};   // 4  размерность вычисляется
int[,] d = new int[2,3] {{1, 2, 3}, {4, 5, 6}};// 5  избыточное описание
```

Рисунок 77 - Прямоугольные массивы

Ступенчатые массивы - особый тип массивов, где количество элементов в разных строках может различаться. В памяти ступенчатый массив хранится иначе, чем прямоугольный - в виде нескольких внутренних массивов, каждый из которых имеет свой размер.

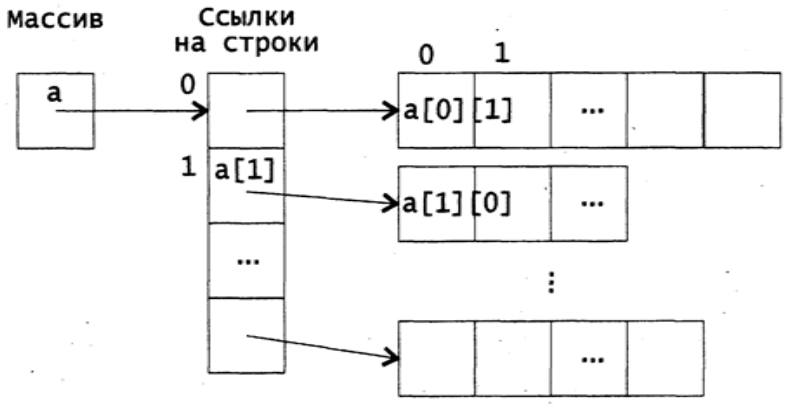


Рисунок 78 - Ступенчатые массивы

Описание ступенчатого массива: тип [ ] [ ] имя; . При этом для каждого из отдельных массивов, составляющих ступенчатый, память необходимо выделять явным образом, например:

```
int [ ][ ] a = new int[3][ ];  // выделение памяти под ссылки на три строки
a[0] = new int[5];             // выделение памяти под 0-ю строку (5 элементов)
a[1] = new int[3];             // выделение памяти под 1-ю строку (3 элемента)
a[2] = new int[4];             // выделение памяти под 2-ю строку (4 элемента) .
```

Ниже представлены члены базового класса Array (таблица 35). В том числе, например: метод Array::Clear(ДескрипторМассива, 0, ДлинаМассива); методы сортировки элементов массива Array::Sort( ) и Array:: Sort (ДескрипторМассива, НачальныйЭлемент, КонечныйЭлемент); метод двоичного поиска элемента массива метод int Array:: BinarySearch(ДескрипторМассива, Образец); и др.

Таблица 35 - Состав класса Array

Элемент	Вид	Описание
Length	Свойство	Количество элементов массива (по всем размерностям)
Rank	Свойство	Количество размерностей массива
BinarySearch	Статический метод	Двоичный поиск в отсортированном массиве
Clear	Статический метод	Присваивание элементам массива значений по умолчанию
Copy	Статический метод	Копирование заданного диапазона элементов одного массива в другой массив
CopyTo	Метод	Копирование всех элементов текущего одномерного массива в другой одномерный массив
GetValue	Метод	Получение значения элемента массива
IndexOf	Статический метод	Поиск первого вхождения элемента в одномерный массив
LastIndexOf	Статический метод	Поиск последнего вхождения элемента в одномерный массив
Reverse	Статический метод	Изменение порядка следования элементов на обратный
SetValue	Метод	Установка значения элемента массива
Sort	Статический метод	Упорядочивание элементов одномерного массива

Примеры работы с массивами приведены ниже в синтаксисе CLI:

```
array<int>^ values = { 3, 5, 6, 8, 6 };
for each (int item in values)
{
   item = 2*item + 1;
   Console::Write("{0,5}", item);
}

int main(array<System::String ^> ^args)
{
   array<double>^ samples = gcnew array<double>(50);

   // Генерировать случайные значения элементов
   Random^ generator = gcnew Random;
   for(int i = 0; i < samples->Length ; i++)
      samples[i] = 100.0*generator->NextDouble();

   Console::WriteLine(L"Массив содержит следующие значения:");
   for(int i = 0; i < samples->Length ; i++)
   {
      Console::Write(L"{0,10:F2}", samples[i]);
      if((i+1)%5 == 0)
         Console::WriteLine();
   }

   double max(0);
   for each(double sample in samples)
      if(max < sample)
         max = sample;
}
Console::WriteLine(L"Максимальное значение = {0:F2}", max);
return 0;
}  .
```

3 Работа со строковыми данными

Тип данных String, предназначенный для работы со строками символов в кодировке Unicode, является встроенным типом языка C#. Ему соответствует базовый класс System.String библиотеки .NET. Создать строковую переменную можно несколькими способами, например:

```
string s; // инициализация строки отложена
string t = "qqq"; // инициализация строки строковым литералом
string u = new string(' ', 20); // конструктор создает строку из 20 пробелов
char[ ] a = { '0', '0', '0' }; string v = new string(a); // инициализация массивом символов.
```

Для строк перегружены операции присваивания, проверки на равенство или неравенство, выбора символа по номеру, сцепления (конкатенации) и др. При этом на равенство проверяются не ссылки, а значения строк. Обращаться к отдельному элементу строки по индексу можно только для получения значения, но не для его изменения. Это связано с тем, что строки типа string относятся к так называемым неизменяемым типам данных. Методы, изменяющие содержимое строки, на самом деле создают новую копию строки. При этом неиспользуемые копии строк автоматически удаляются сборщиком мусора. Для перевода числовых данных в строковые можно использовать метод ToString(). Например

Int32 Value = 100; String hStr; hStr = Value.ToString( ); .

Обратный перевод может быть выполнен как

Value = Int32.Parse(hStr); или Value = System.Convert.ToInt32(hStr) .

Примеры работы со строковыми данными приведены ниже в синтаксисе CLI:

```
int main(array<System::String ^> ^args)
{
   array<String^>^  names = { "Jill", "Ted", "Mary", "Eve", "Bill",
                        "Al", "Ned", "Zoe", "Dan", "Jean"};
   array<int>^  weights = { 103, 168, 128, 115, 180,
                       176, 209, 98, 190, 130 };
   array<String^>^  toBeFound = {"Bill", "Eve", "Al", "Fred"};

   Array::Sort(names, weights); // Сортировка массива
   int result = 0;              // Хранит значение возврата
   for each(String^ name in toBeFound) // Поиск весов
   {
      result = Array::BinarySearch(names, name); // Поиск в массиве имен
      if(result < 0) // Проверка результата поиска
         Console::WriteLine(L"{0} не найден.", name);
      else
         Console::WriteLine(L"{0} весит {1} фунтов.", name,
                        weights[result]);
   }
   return 0;
}

int main(array<System::String ^> ^args)
{
   array<wchar_t>^ punctuation = {L'\'', L'\"', L'.', L'.', L'.', L'.', L',', L'!', L'?', L'?'};
   String^ sentence(L"\"It's chilly in here\", the boy's mother said coldly.");

   // Создать массив пробелов длиной sentence
   array<wchar_t>^ indicators(gcnew array<wchar_t>(sentence->Length) {L' '});

   int index(0); // Индекс найденного символа
   int count(0); // Счётчик знаков препинания
```

```
    while((index = sentence->IndexOfAny(punctuation, index)) >= 0)
    {
        indicators[index] = L'^'; // Установим маркер
        ++index; // Инкремент до следующего символа
        ++count; // Увеличить счётчик
    }
    Console::WriteLine(L"Найдено {0} знаков препинания в строке:", count);
    Console::WriteLine(L"\n{0}\n{1}", sentence, gcnew String(indicators));

    return 0;
}  .
```

Лекция № 15 Ссылочные типы

1 Классы

Состав классов представлен ниже (рисунок 79). Синтаксис описания класса:

[атрибуты] [спецификаторы] class имя_класса [: предки] { // тело_класса } .

Синтаксис описания атрибута:

[атрибуты] [спецификаторы] [const] тип имя [= начальное_значение] .

Синтаксис описания метода:

[атрибуты] [спецификаторы] тип имя_метода ([параметры]) { // тело_метода }

Для инициализации объектов предназначены конструкторы классов.

Конструктор вызывается автоматически при создании объекта класса с помощью операции new. Имя конструктора совпадает с именем класса. Конструктор не возвращает значений.

Класс может иметь несколько конструкторов с разными параметрами для разных видов инициализации.

При отсутствии конструкторов автоматически полям значимых типов присваивается нуль соответствующего типа, полям ссылочных типов - значение null.

Пример описания класса (в синтаксисе CLI это класс-ссылка) и использования конструкторов приведен ниже.

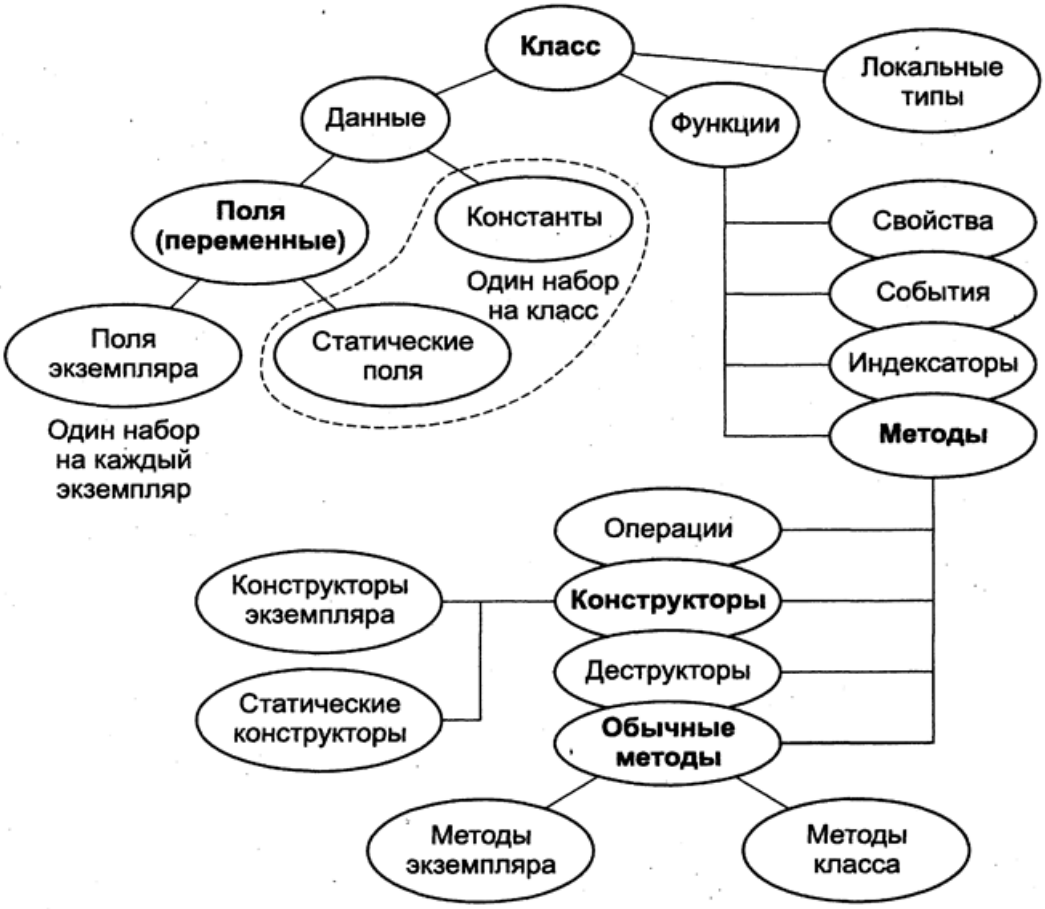


Рисунок 79 - Состав классов

```
ref class Box
{
public:
   // конструктор без параметров
   Box(): Length(1.0), Width(1.0), Height(1.0)
   {
      Console::WriteLine(L"Вызван конструктор без параметров");
   }

   // конструктор с параметрами
   Box(double lv, double bv, double hv):
      Length(lv), Width(bv), Height(hv)

{
   Console::WriteLine(L"Вызван конструктор с параметрами");
}

// вычисление объёма ящика
double Volume()
{
   return Length*Width*Height;
}
```

```
private:
   double Length;    // Длина ящика
   double Width;     // Ширина ящика
   double Height;    // Высота ящика
};

int main(array<System::String ^> ^args)
{
   Box^ aBox;
   Box^ newBox = gcnew Box(10, 15, 20);
   aBox = gcnew Box; // Инициализировать по умолчанию
   Console::WriteLine(L"Объём по умолчанию: {0}", aBox->Volume());
   Console::WriteLine(L"Объём нового ящика: {0}", newBox->Volume());
   return 0;
} .
```

Для упрощения работы с классами в них используют компоненты – свойства. Свойства служат для организации доступа к атрибутам класса. Как правило, свойство связано с закрытым полем класса и определяет методы его получения и установки. Синтаксис свойства:

```
[ атрибуты ] [ спецификаторы ] тип имя_свойства {   [ секция get код_доступа ]
   [ секция set код_доступа ] } .
```

Значения спецификаторов, используемых для свойств и методов аналогичны. Чаще всего свойства объявляются как открытые (со спецификатором public), поскольку они входят в интерфейс объекта. Пример добавления свойства показан ниже

```
public class Button: Control
{
    private string caption;      // закрытое поле, с которым связано свойство
    public string Caption {      // свойство
       get {                     // способ получения свойства
          return caption;
       }
       set {                       // способ установки свойства
          if (caption != value) {
             caption = value;
          }
       }
    }
    ...
}
```

Пример добавления свойства в синтаксисе CLI показан ниже

```
ref class Weight
{
private:    int lbs;    int oz;
public:
   property int pounds
   {
```

```
        int get() { return lbs; }
        void set(int value) { lbs = value; }
    }

    property int ounces
    {
        int get() { return oz; }
        void set(int value) { oz = value; }
    }
};  .
```

Здесь свойства pounds и ounces используются для предоставления доступа к значениям закрытых полей lbs и oz. Например, можно установить значения свойств объекта Weight и затем обращаться к ним так, как показано ниже:

```
Weight^ wt = gcnew Weight; wt->pounds = 162; wt->ounces = 12;
Console::WriteLine(L"Вес равен {0} фунтов {1} унций.", wt->pounds, wt->ounces); .
```

2 Особенности использования методов классов

Синтаксис метода:

[атрибуты] [спецификаторы] тип имя_метода ([параметры]) { // тело_метода } .

В C# для обмена данными с вызываемым методом предусмотрено четыре типа параметров: - параметры-значения (передача копий как локальных переменных); - параметры-ссылки (передача адресов), описываемые спецификатором ref; - выходные параметры (только для возврата значений), описываемые спецификатором out; - параметры-массивы, описываемые спецификатором params. Здесь базовые способы – передача значений параметров по значению или по ссылке. При передаче по значению метод получает копии значений аргументов. Доступа к исходным значениям аргументов у метода нет как возможности их изменить. При передаче по ссылке (по адресу) метод получает копии адресов аргументов и при необходимости может изменять исходные значения аргументов.

Ниже приведены примеры описания прототипов методов и их сигнатур:

- метод для обмена данных между переменными x, y

```
static public void TO_SWAP(ref int x, ref int y)   ;
```

- метод для получения целой Whole и дробной Frac частей числа Number

```
static public void TO_GET_PARTS(double Number, out int Whole, out double Frac)  ;
```

- метод для получения целой Whole и дробной Frac частей числа Number (исходное число обнулить)

```
static public void TO_GET_PARTS(ref double Number, out int Whole, out double Frac)  ;
```

- метод с переменным числом параметров для вывода комментария Caption и результатов обработки значений Numbers

```
static public int MIN_VALUE(string Caption, params int [ ] Numbers)  ;
```

- метод для расчета числа положительных значений NumberOfPositive в массиве Numbers и возврата преобразованного массива Numbers

```
static public int[ ] MODIFIED_ARRAY(int [ ] Numbers, out int NumberOfPositive)  ;
```

- метод для обмена значений двух объектов класса ABOUT_SWAP

static public void TO_SWAP(ref ABOUT_SWAP rObject1, ref ABOUT_SWAP rObject2) .

3 Структуры

Структура — тип данных, аналогичный классу, но имеющий ряд важных отличий от него: - структура является значимым, а не ссылочным типом данных, то есть экземпляр (объект) структуры хранит значения своих элементов, а не ссылки на них, и располагается в стеке, а не в куче; - структура не может участвовать в иерархиях наследования, но может реализовывать интерфейсы; - в структуре запрещено определять конструктор по умолчанию, поскольку он определен неявно и присваивает всем ее элементам значения по умолчанию (нули соответствующего типа); - в структуре запрещено определять деструкторы, поскольку это бессмысленно.

Отличия от классов обусловливают область применения структур: это типы данных, имеющие небольшое количество полей, с которыми удобнее работать как со значениями, а не как со ссылками. Накладные расходы на динамическое выделение памяти для небольших объектов могут весьма значительно снизить быстродействие программы, поэтому их эффективнее описывать как структуры, а не как классы. С другой стороны, передача структуры в метод по значению требует дополнительного времени и дополнительной памяти.

Синтаксис описания структуры:

[атрибуты] [спецификаторы] struct имя_структуры [: интерфейсы]
{ тело_структуры [:] } .

Интерфейсы, реализуемые структурой, перечисляются через запятую. Тело структуры может состоять из констант, полей, методов, свойств, событий, индексаторов, операций и вложенных типов. Правила их описания и использования аналогичны соответствующим элементам классов, за исключением отличий, вытекающих из упомянутых ранее: - так как структуры не могут участвовать в иерархиях, то для их элементов не могут использоваться спецификаторы protected и protected internal - применяются только public, internal и private (последний только для вложенных структур); - структуры не могут быть абстрактными и по умолчанию не наследуются - бесплодны (sealed); - методы структур не могут быть абстрактными и виртуальными, переописывать (со спецификатором override) можно только методы, унаследованные от базового класса object; - параметр this интерпретируется как значение, поэтому его можно использовать для ссылок, но не для присваивания; - при описании структуры нельзя задавать значения полей по умолчанию, так как это будет сделано в конструкторе по умолчанию, создаваемом автоматически. Пример описания структуры приведен ниже в синтаксисе CLI (там это класс-значение):

```
// рост
value class Height
{
   private:
      // рост в футах и дюймах
      int feet; }
int inches;
public:
```

```
// рост в дюймах
Height(int ins)
{
   feet = ins/12;
   inches = ins%12;
}
// рост в футах и дюймах
Height(int ft, int ins) : feet(ft), inches(ins){}
int main(array<System::String ^> ^args)
{
   Height myHeight = Height(6, 3);
   Height^ yourHeight = Height(70);
   Height hisHeight = *yourHeight;
   Console::WriteLine(L"Мой рост есть {0}", myHeight);
   Console::WriteLine(L"Твой рост есть {0}", yourHeight);
   Console::WriteLine(L"Его рост есть {0}", hisHeight);
   return 0;
}
```

Пример добавления свойства:

```
// Рост в метрах как свойство
property double meters
{
   // Возвращает значение свойства
   double get()
   {
      return inchesToMeters*(feet*inchesPerFoot+inches);
   }

   // Здесь можно определить функцию set()
}  .
```

Лекция № 16 Ссылочные типы

1 Перегрузка операторов (операций)

Операции класса описываются с помощью методов специального вида (методов-операций). Перегрузка операций похожа на перегрузку обычных методов. Синтаксис операций:

[атрибуты] спецификаторы имя_операции тело_операции

Объявитель операции содержит ключевое слово operator, по которому и опознается описание операции в классе. При описании операций необходимо соблюдать следующие правила: - операция должна быть описана как открытый статический метод класса (public static); - параметры передаются в операцию только по значению (нельзя исполь-

зовать спецификаторы ref или out); - сигнатуры всех перегруженных операций класса должны различаться.

Используются три вида операций класса: унарные, бинарные и операции преобразования типа. Унарные операции (+ - ! ~ ++ -- true false и др.) имеют прототипы как показано ниже

```
public static int operator +(MyObject m)
public static MyObject operator --(MyObject m)
public static bool operator true(MyObject m)
```

Параметр, передаваемый в операцию, должен иметь тип класса, для которого она определяется. Операции не должны изменять значение передаваемого им операнда.

Бинарные операции (+ - * / % & | ^ << >> == != > < >= <=) имеют прототипы как показано ниже

```
public static MyObject operator +(MyObject m1, MyObject m2)
public static bool operator ==(MyObject m1, MyObject m2)   .
```

При этом хотя бы один параметр, передаваемый в операцию, должен иметь тип класса, для которого она определяется. Операция может возвращать величину любого типа. Операции выявления отношений перегружаются только парами и обычно возвращают логическое значение. Чаще всего в классе определяют операции сравнения на равенство и неравенство для того, чтобы обеспечить сравнение объектов, а не их ссылок, как определено по умолчанию для ссылочных типов. Примеры перегрузки операторов (в синтаксисе CLI) приведены ниже

```
ref class Length
{
private:
   int feet;
   int inches;

public:
   static initonly int inchesPerFoot = 12;

   // Конструктор
   Length(int ft, int ins) : feet(ft), inches(ins) { }

   // Длина в виде строки
   virtual String^ ToString() override
   {
      return feet.ToString() + (1 == feet ? L" foot " : L" feet ") +
            inches + (1 == inches ? L" inch" : L" inches");
   }

   // оператор сложения
   Length^ operator+(Length^ len)
   {
      int inchTotal = inches + len->inches + inchesPerFoot * (feet + len->feet);
      return gcnew Length(inchTotal / inchesPerFoot, inchTotal % inchesPerFoot);
```

```
    }

    // оператор деления - правый операнд типа double
    static Length^ operator/(Length^ len, double x)
    {
        int ins = safe_cast<int>((len->feet * inchesPerFoot + len->inches) / x);
        return gcnew Length(ins / inchesPerFoot, ins % inchesPerFoot);
    }
};
// оператор деления - оба операнда типа Length
static int operator/(Length^ len1, Length^ len2)
{
    return (len1->feet * inchesPerFoot + len1->inches) /
           (len2->feet * inchesPerFoot + len2->inches);
}

// оператор остатка от деления
static Length^ operator%(Length^ len1, Length^ len2)
{
    int ins = (len1->feet * inchesPerFoot + len1->inches) %
              (len2->feet * inchesPerFoot + len2->inches);
    return gcnew Length(ins / inchesPerFoot, ins % inchesPerFoot);
}

static Length^ operator*(double x, Length^ len); // Умножение - справа double
static Length^ operator*(Length^ len, double x); // Умножение - слева double

// Префиксный и постфиксный операторы инкремента
static Length^ operator++(Length^ len)
{
    temp->feet += temp->inches / temp->inchesPerFoot;
    temp->inches %= temp->inchesPerFoot;
    return temp;
};

// оператор умножения - правый операнд double
Length^ Length::operator*(double x, Length^ len)
{
    int ins = safe_cast<int>(x * len->inches + x * len->feet * inchesPerFoot);
    return gcnew Length(ins / inchesPerFoot, ins % inchesPerFoot);
}

// оператора умножения - левый операнд double
Length^ Length::operator*(Length^ len, double x)
{
    return operator*(x, len);
}
```

2 Наследование классов

При наследовании классов конструкторы не наследуются, поэтому производный класс должен иметь собственные конструкторы. Порядок вызова конструкторов определяется приведенными далее правилами: - если в конструкторе производного класса явный вызов конструктора базового класса отсутствует, то автоматически вызывается конструктор базового класса без параметров; - для иерархии классов, состоящей из нескольких уровней, конструкторы базовых классов вызываются, начиная с самого верхнего уровня и каждый конструктор инициализирует свою часть объекта; - если конструктор базового класса требует указания параметров, он должен быть явным образом вызван в конструкторе производного класса в списке инициализации с помощью ключевого слова base как показано в примере ниже.

```
public Demo(int a) : base()
{
   this.a = a;
}  .
```

Поля, методы и свойства класса наследуются, поэтому при желании заменить элемент базового класса новым элементом следует явным образом сообщить это компилятору с помощью ключевого слова new. Если в производном классе требуется переопределить виртуальный метод, используется ключевое слово override, например

```
override public void Passport() { ... }  .
```

Переопределенный виртуальный метод должен обладать таким же набором параметров, как и одноименный метод базового класса. Это требование вполне естественно, если учесть, что одноименные методы, относящиеся к разным классам, могут вызываться из одной и той же точки программы.

Возможно использование абстрактных классов, которые нужны только для наследования – создания производных классов (потомков). Как правило, в абстрактном классе задается набор методов, которые каждый из потомков может реализовать по-своему. При этом в нем используются методы без описания тел, которые объявляются со спецификатором abstract. Таким образом, абстрактные классы задают общие требования к классам иерархии (интерфейсам), которые предполагается конкретизировать в производных классах. Пример описания абстрактного класса в синтаксисе CLI (класс используется далее в роли базового класса) приведен ниже

```
// Абстрактный базовый класс - Container
ref class Container abstract
{
public:
   virtual double Volume() abstract;
   virtual void ShowVolume()
   {
      Console::WriteLine(L"Объём равен {0}", Volume());
   }
};  .
```

Пример организации наследования в синтаксисе CLI приведен ниже

```
ref class Box : Container  // производный класс
{
public:
   virtual void ShowVolume() override
   {
      Console::WriteLine(L"Полезный объём Box = {0}", Volume());
   }

   virtual double Volume() override
   { return m_Length*m_Width*m_Height; }

   Box() : m_Length(1.0), m_Width(1.0), m_Height(1.0) {}
   Box(double lv, double wv, double hv):
      m_Length(lv), m_Width(wv), m_Height(hv) { }
protected:
   double m_Length;
   double m_Width;
   double m_Height;
};   .
```

3 Интерфейсы

Интерфейс (интерфейсный класс) напоминает абстрактный класс. В нем задается набор открытых абстрактных статических членов - методов, свойств, индексаторов, которые должны быть реализованы в производных классах. Таким образом, интерфейс определяет поведение, которое поддерживают реализующие этот интерфейс классы. Соответственно к членам (методам) объектов таких классов можно обращаться одинаковым образом, а реализация элементов интерфейса уникальна в каждом производном классе. Указанное использует механизм динамического полиморфизм, когда объекты разных классов по-разному реагируют на вызовы одного и того же метода (говорят - реакция зависит от контекста вызова!). Синтаксис интерфейса аналогичен синтаксису класса:

```
[ атрибуты ] [ спецификаторы ] interface имя_интерфейса [ : предки ]
   тело_интерфейса [ ; ]
```

Интерфейс определяется с помощью ключевых слов interface class или interface struct. Независимо от того, использованы указанные ключевые слова для определения интерфейса или нет, все его члены являются по умолчанию открытыми.

Могут использоваться спецификаторы new, public, protected, internal и private. Спецификатор new применяется для вложенных интерфейсов и имеет такой же смысл, как и соответствующий модификатор метода класса. Остальные спецификаторы управляют видимостью членов интерфейса. В разных контекстах определения интерфейса допускаются разные спецификаторы. По умолчанию интерфейс доступен только из текущей сборки, в которой он описан (спецификатор internal).

Интерфейс может наследовать свойства нескольких интерфейсов, в этом случае предки перечисляются через запятую (допустимо одиночное и множественное прямое наследование интерфейсов). Соответственно конкретный интерфейсный класс может

быть производным по отношению к другим интерфейсным классам и использовать все члены перечисленных интерфейсов. Пример описания наследования интерфейсов

```
interface class IController : ITelevision, IRecorder
    // члены IController
```

Здесь интерфейс IController включает в себя собственные члены, а также наследует члены интерфейсов ITelevision и IRecorder. В классе, который реализует интерфейс IController, должны быть определены функции-члены из интерфейсов IController, ITelevision и IRecorder. Ниже приведен пример описания интерфейса и класса на его основе в синтаксисе языка CLI

```
interface class IContainer
{
    double Volume();
    void ShowVolume();
};
ref class Box : IContainer
{
public:
    virtual void ShowVolume()
    {
        Console::WriteLine(L"Полезный объём Box равен {0}", Volume());
    }
    virtual double Volume()
    {
        return m_Length * m_Width * m_Height;
    }
    Box() : m_Length(1.0), m_Width(1.0), m_Height(1.0) { }
    Box(double lv, double wv, double hv) :
    m_Length(lv), m_Width(wv), m_Height(hv) { }
protected:
    double m_Length;
    double m_Width;
    double m_Height;   };   .
```

2 ПРАКТИЧЕСКИЙ РАЗДЕЛ

Методические указания к выполнению лабораторных работ

ЛАБОРАТОРНАЯ РАБОТА № 1 “Типовой каркас оконного windows-приложения (ТКП). Обработка сообщений. Организация вывода в клиентскую область окна”

ЧАСТЬ 1

ЦЕЛЬ РАБОТЫ: 1. Ознакомиться со структурой и особенностями программирования оконных windows-приложений. 2. Изучить каркасы windows-приложений. 3. Изучить структуру типового каркаса оконных приложений (ТКП), использование ТКП для создания пользовательских приложений.

СОСТАВ ОТЧЕТА: 1. Описание каркасов windows-приложений (Empty, Simple или аналогичных): - характеристика интерфейса, предоставляемых возможностей; - файловый состав (структура проекта, дерево папок, состав, назначение файлов и их соподчиненность по включению); - функциональный состав (схема иерархии функций приложения, назначение и прототипы функций). 2. Аналогичное описание ТКП. Диаграммы прецедентов, состояний, компонентов. 3. Выводы: а) о последовательности инициализации ТКП; б) о причинах и условиях посылки сообщений ТКП, рассмотренных в работе; в) перечень ситуаций “перерисовки”; г) выводы по способу завершения работы приложения.

ЗАДАНИЯ ДЛЯ ВЫПОЛНЕНИЯ. Все задания выполняются в системе Visual Studio.

Задание 1. Изучить теоретический материал об особенностях оконных приложений (лекция 1, параграфы “Особенности программирования в ОС Windows”, “Особенности Windows-приложений на языках C, C++” и др.). Обратить внимание на понятие ”оконное” риложение, структуру приложения (лекция 2, параграфы “Особенности оконных приложений Windows”, “Система Visual Studio. Каркасное программирование”).

Задание 2. Создать каркас windows-приложения (тип Empty или аналогчный) средствами мастера (например, Win32 Application). Изучить его свойства (состав интерфейса; файловый состав; функциональный состав). Для этого в системе Visual Studio создать новый проект (New Project): - выбрать пункт меню File-New, в появившемся диалоговом окне выбрать вкладку проекты Projects и выбрать пункт Win32 Application, т.е. создание windows-приложения; - набрать имя проекта в строке Project name, установить переключатель типа проекта в An empty project (пустой проект), завершить создание каркаса (дерево проекта можно увидеть в окне рабочего пространства – вкладка FileView !) и запустить приложение.

Задание 3. Создать проект из одного файла с одной, главной функцией, выводящей сообщения с помощью функций MessageBox. Для этого доработать каркас - создать файл в папке Source Files проекта приложения: - выбрать пункт меню File-New, в появившемся диалоговом окне выбрать вкладку файлы Files и пункт C++ Source File, т.е. создание файла с текстом на языке C++; - набрать имя файла в поле File name и завершить его создание; - включить в созданный файл заголовок #include <windows.h>, создать пустую функцию WinMain, например, как показано ниже и выполнить приложение

```
int  WINAPI  WinMain (HINSTANCE H1, HINSTANCE H2, LPSTR Str, int I)
{   return 0;  }   ;
```

- включить в приложение вывод окна сообщения MessageBox (лекция 3, параграф “Преобразование типов, ввод-вывод данных”). При необходимости исправить неточности! Соответствующую справочную информацию можно получить по F1 в параграфе CWindow::MessageBox – Microsoft Foundation classes …, либо выделить строку – MessageBox и нажать клавишу F1. При этом, как правило, появится вкладка ”найденные разделы”, где следует уточнить, что ищется. В данном случае это API-функция. При работе с классами это может быть CWnd::MessageBox в разделе MFC and Template… . Функция MessageBox имеет прототип

РезультатВыбора MessageBox (ДескрипторРодительскогоОкна, Сообщение, НазваниеОкнаСообщения, СоставКнопокОкнаСообщения) .

Здесь состав кнопок задается кодами MB_OK, MB_OKCANCEL, MB_YESNOCANCEL и другими или их числовыми эквивалентами, а тип возвращаемого результата зависит от нажатой кнопки и кодируется ее дескриптором как IDYES, IDOK, IDCANCEL и т.д. Например, включить последовательно

```
MessageBox ( NULL, "Работает приложение", "Сообщение1", 1),
MessageBox (hWnd, "Работает приложение", "Сообщение2", MB_OK),
MessageBox (hWnd, "Работает приложение", " Сообщение3", MB_YESNOCANCEL) .
```

Проанализировать возвращаемый код для одного из MessageBox (с двумя и более кнопками) – выводить сообщение о типе нажатой кнопки.

Задание 4. Создать проект из двух файлов – главный (с расширением cpp) с текстом программы - функцией WinMain, заголовочный (с расширением h) с командами препроцессора. Для этого повторить задание 3, модифицировать приложение, заголовочный файл подключить к главному командой препроцессора #include "*.h". Выполнить приложение.

Задание 5. Создать каркас Windows-приложения (тип Simple или аналогичный) средствами мастера (аналогично заданию 2). Изучить его свойства: интерфейс; - файловый состав; - функциональный состав; - ресурсный состав.

Задание 6. Создать проект, модифицировав созданный каркас путем вставки в WinMain функций MessageBox (аналогично заданию 3).

Задание 7. Изучить теоретический материал о сообщениях и типовом каркасе приложения (лекция 4, разделы “Сообщения. Обработчики сообщений - Общая схема обработки сообщений”, лекция 5, разделы “Структура ТКП - Структура программного обеспечения ТКП”, “ТКП. Создание в системе Visual Studio”).

Создать приложение на базе типового каркаса приложения ТКП, взяв за основу автоматический каркас типа Simple (или аналогичный). Изучить его свойства: - интерфейс; - файловый состав; - функциональный состав; - ресурсный состав. Для этого: - создать каркас проекта, тип Simple (см. задание 5); - в основной файл проекта (*.cpp) вместо имеющегося там текста вставить текст ТКП (см. лекция 5, раздел “ТКП. Создание в системе Visual Studio”), сохранив только команду #include "stdafx.h"; - выполнить приложение.

Задание 8. Модифицировать приложение, последовательно размещая MessageBox с сообщением о выполненной функции: а) после создания окна в CreateWindow; б) после вывода и после обновления окна UpdateWindow (hWnd); в) сразу в секции case WM_DESTROY; г) в секции case WM_PAINT в место для вставки фрагмента пользователя. Для каждого варианта вставки осуществить запуск приложения и рассмотреть его

работу (вывод сообщений) при свертывании-развертывании окна, перекрытии, перемещении окна, попытке изменения размеров окна.

Закомментировать case WM_DESTROY: PostQuitMessage(0); . Выполнить несколько запусков ТКП и убедиться, что после закрытия приложений – они по-прежнему активны – но без окон – присутствуют в списке процессов компьютера!!! Разработать автоматную диаграмму приложения.

Задание 9. Повторить создание приложения (задание 7). Для этого: - создать каркас проекта, тип Simple (см. задание 5); - в основной файл проекта (*.cpp) вместо имеющегося там текста вставить текст ТКП (см. лекция 5, раздел “ТКП. Создание в системе Visual Studio”), сохранив только команду #include "stdafx.h"; - выполнить приложение и при необходимости устранить ошибки, вызванные присутствием в тексте недоопределенных элементов. Вставить функции MessageBox.

Задание 10. Изучить теоретический материал по созданию окна приложения (см. лекция 6, раздел “Создание класса (стиля) окна. Структура WNDCLASS”, “Создание и визуализация окна”). Создать приложение, модифицировав приложение на базе ТКП. Изменить его интерфейс (вид окна, состав элементов окна - кнопок и т.п.). Для этого: - создать приложение на базе ТКП (см. задание 7); - выполнить отмену опции окна - WS_OVERLAPPEDWINDOW и обеспечить тот же состав окна с применением опции окна - WS_OVERLAPPED путем поэлементного включения соответствующих элементов (кнопок и т.п.) через оператор “или”; - выполнить отмену опции окна ТКП - WS_OVERLAPPED и обеспечить тот же состав окна, но путем поэлементного включения соответствующих элементов (кнопок и т.п.).

Внести изменения в параметры: - WNDCLASS; - функции CreateWindow. Например, добавить в окно рамки, полосы прокрутки, убрать или добавить командные кнопки, внести изменения в координаты и размеры окна, использовав соответствующие параметры функции (x, y, Width, Height); - функции ShowWindow.

Задание 11. Обеспечить попеременное переключение окна – “спрятано”, “показано”, “свернуто”, “развернуто”.

КОНТРОЛЬНЫЕ ЗАДАНИЯ

1. Создать приложение с ТКП “вручную”, не используя мастер, поменяв размеры окна (200 на 200), заголовок окна, добавив полосы прокрутки. 2. Добавить к приложению обработчик сообщения WM_QUIT – осуществить вывод окна сообщения MessageBox о пришедшем сообщении. 3. Повторить задание 1 в заданной версии Visual Studio. 4. Обеспечить попеременное переключение окна ТКП (“спрятано”-“показано”), использовав для этого обработчик сообщения WM_LBUTTONDOWN - нажатие левой клавиши “мыши” или нажатие клавиши на клавиатуре.

ЧАСТЬ 2

ЦЕЛЬ РАБОТЫ: 1. Ознакомиться с особенностями организации вывода в оконных windows-приложениях. 2. Ознакомиться с особенностями управления сообщениями.

СОСТАВ ОТЧЕТА: Описание приложений (задания 9-11): - состав интерфейса и его возможности; - файловый состав (структура проекта, дерево папок, состав, назначение файлов и их соподчиненность по включению); - функциональный состав (привести схему иерархии функций приложения, описать назначение и прототипы функций). Диаграммы прецедентов, состояний, компонентов, классов. Соответствующие листинги.

ЗАДАНИЯ ДЛЯ ВЫПОЛНЕНИЯ. При разработке приложений следует учитывать: 1. Особенности использования контекста устройства (КУ - HDC hdc) - системного ресурса для хранения атрибутов объектов, связанных с рисованием. Используется в приложени-

ях для вызова функций GDI. Как правило, КУ выделяется только в одном месте программы. Например, создание hdc = BeginPaint(hWnd, &ps) и разрушение EndPaint(hWnd, ps). 2. Особенности ввода-вывода данных, где, как правило, используется строковый тип. Для преобразований типов данных применяют любые подходящие функции, например sscanf, sprintf из библиотек С, или аналогичные функции (см. лекция 3, параграф “Преобразование типов данных, ввод-вывод данных”).

Задание 1*. Создать ТКП, описать схему иерархии модулей. Выполнить контрольные задания.

Задание 2. Изучить теоретический материал (см. лекция 3, параграф “Преобразование типов, ввод-вывод данных”).

Задание 3. Создать приложение на базе ТКП. Добавить строку приветствия (координаты вывода - 0, 0), для чего вставить в обработчик WM_PAINT пользовательский фрагмент

```
char   szText [25] =  “Работает ТКП” ;
…….
TextOut ( hdc, 0, 0, szText, strlen ( szText ) );   .
```

Организовать вывод данных, например,

```
int   X = 2007;  char  szXasString[100];
……
wsprintf ( szXasString, "Значение X - %d", X );
TextOut ( hdc, 0, 130, szXasString, strlen ( szXasString ) );  .
```

Вывести в столбик для значений X от 1 до 10 (с шагом 0,5) пары значений: X - квадрат X. Запустить приложение. Выполнить свертывание-развертывание окна, перемещение, изменение размеров окна. Убедиться в автоматической перерисовке окна.

Задание 4. Модифицировать предыдущее приложение. Для этого: - изменить формат вывода и увеличить число обрабатываемых чисел до 75; - добавить вывод номера перерисовки (переменная int ReDrawNumber = 0). Вид окна приведен на рисунке 80 ниже.

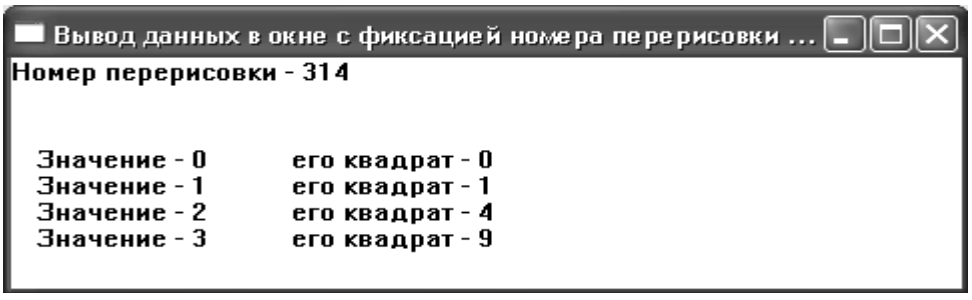


Рисунок 80 - Фрагмент интерфейса

Задание 5*. Создать приложение для табулирования заданной функции. Значения вывести в виде таблицы. Параметры – левая, правая границы, число точек табулирования.

Задание 6*. Модифицировать предыдущее приложение, организовав вычисления на базе пользовательских классов.

Задание 7. Создать приложение на базе ТКП. Организовать вывод графических данных. Основные графические примитивы рисуются с использованием библиотечных функций: - дуги эллипса Arc, ArcTo; - эллипсы и окружности Ellipse; - линии от текущей

точки до точки, указанной в функции LineTo(hdc, x, y); - прямоугольники Rectangle; - полигоны Polygon; - связанные отрезки прямых PolyLine. Также используют функцию рисования точки COLORREF SetPixel (hdc, x, y, COLORREF crColor), функцию перемещения курсора к указанной точке с сохранением координаты текущей точки в структуре типа LPPOINT - BOOL MoveToEx(hdc, x, y, LPPOINT pPoint). Фрагмент использования функций приведен ниже

```
                                        ...
case WM_PAINT:
        hdc = BeginPaint( hWnd, &ps );
        hPen = CreatePen(PS_SOLID,1,RGB (0, 255, 0) );
        SelectObject (hdc, hPen);
        Ellipse ( hdc, 100, 100, 300, 300 );
        MoveToEx(hdc, 50, 50, NULL);
        LineTo(hdc, 400,400);
          ……..
        ValidateRect(hWnd,NULL);
        EndPaint(hWnd, &ps);
break;   ...
```

Задание 8. Создать приложение на базе ТКП. Включить чувствительность к нажатию левой клавиши “мыши” (сообщение WM_LBUTTONDOWN) - выводить окно MessageBox. Для этого создать в функции-обработчике дополнительную секцию

```
case WM_LBUTTONDOWN:
      // < ФРАГМЕНТ ПОЛЬЗОВАТЕЛЯ >
break;        ,
```

аналогичную, например,

```
case WM_PAINT:    ……    break;    .
```

Повторить задания 3-4, выполняя все действия в секции case WM_LBUTTONDOWN (секция case WM_PAINT должна быть пустой). Запустить приложение. Выполнить свертывание-развертывание окна, перемещение, перекрытие, изменение размеров окна. Проанализировать характер перерисовки содержимого окна.

Задание 9. Создать приложение, в котором подсчитываются события - нажатие левой клавиши мыши. А результат - число нажатий - выводится в секции WM_PAINT только по нажатии правой клавиши мыши. Разработать автоматную диаграмму.

Задание 9.1. Модифицировать приложение. Результат выводить в секции WM_PAINT сразу при его обновлении. Для этого инициировать перерисовку окна функцией InvalidateRect (hWnd, NULL, TRUE). Разработать автоматную диаграмму.

Задание 10. Создать приложение для вывода текста “Работает ТКП”, начиная с позиции X, Y. За начальное значение координат взять 0, 0. По сообщению WM_LBUTTONDOWN увеличивать координаты точки вывода на 50 единиц и инициировать перерисовку. По сообщению WM_RBUTTONDOWN уменьшать координаты точки вывода на 50 единиц и инициировать перерисовку. Разработать автоматную диаграмму.

Задание 11. Создать приложение для вывода координат указателя “мыши” в виде X = …, У = ….. За начальное значение координат взять 0, 0. По сообщению WM_LBUTTONDOWN считывать текущие координаты курсора мыши. По сообщению WM_RBUTTONDOWN инициировать вывод координат. Обеспечивать перерисовку окна.

Блокировать обновление вывода по другим причинам перерисовки. Разработать автоматную диаграмму.

КОНТРОЛЬНЫЕ ЗАДАНИЯ

1. Создать приложение на базе ТКП. Выводить символ, задаваемый с клавиатуры, в координатах указателя “мыши”, задаваемых “щелчком” ее левой клавиши. Начальное значение координат взять как X = 0, Y = 0. Обеспечивать перерисовку и обновление окна как при изменении координат так и при вводе нового символа. Для хранения данных использовать иерархию классов ДАННЫЕ, СИМВОЛ, КООРИНАТА. Предварительно разработать и представить диаграмму классов и диаграмму состояний приложения! 2. Создать windows-приложение и повторить задание 3, использовав при выводе данные реальной длины строк, а также данные об интервалах между строками.

ЛАБОРАТОРНАЯ РАБОТА № 2 “Создание интерфейса оконного windows-приложения (окна, диалоговые окна). Автоматический каркас приложения”

ЦЕЛЬ РАБОТЫ. 1. Изучить создание Windows-приложений с использований каркаса Hello (или аналогичного каркаса). 2. Изучить технологию работы с диалоговыми окнами и элементами управления (ЭУ). Применением диалогового окна в составе главного, в качестве главного окна. 3. Изучить использование кнопок, окон редактирования.

СОСТАВ ОТЧЕТА. 1. Описание каркаса Hello: - состав интерфейса; - файловый состав (структура проекта, папок, состав, назначение файлов и их соподчиненность по включению); - функциональный состав (схема иерархии функций приложения, назначение и прототипы функций). Диаграммы состояний, компонентов. 2. По заданиям 5, 6: - диаграммы состояний; листинги. 3. По заданиям 8, 11: - диаграммы состояний.

ЗАДАНИЯ ДЛЯ ВЫПОЛНЕНИЯ.

Задание 1*. Выполнить контрольные задания.

Задание 2. Изучить теоретический материал - каркас Hello (лекция 2, параграф “Система VISUAL STUDIO. Каркасное программирование”).

Задание 3. Создать каркас Hello средствами мастера. Изучить его свойства: - интерфейс; - файловый, функциональный, ресурсный состав.

Задание 4. Создать каркас. Изменить интерфейс приложения: - изменить названия пунктов меню (вкладка Resource); - поменять название главного окна на свою фамилию; - изменить текст в окне-справке; - по нажатию в окне-справке кнопки ОК выводить новое окно MessageBox с сообщением выхода из окна-справки; - по завершении работы с приложением выводить окно MessageBox с соответствующим предупреждением.

Задание 5. Повторить задание 9 лабораторной работы № 1, часть 2 “Разработка приложений на базе ТКП. Организация вывода”.

Задание 6. Создать приложение для вывода в клиентское окно массива строк ("Use", "function", "prototype","Get","Text","ExtentPoint()") в виде

Use
function prototype
GetTextExtentPoint( )

Для этого использовать данные реальной длины строк и данные об интервалах между строками.

Задание 7. Изучить теоретический материал по работе с графическими ресурсами приложений (лекция 7, параграф " Редакторы ресурсов. Создание ресурсов").

Задание 8. Использование диалогового окна в составе главного. *Создать приложение, при запуске которого в качестве главного выводится масштабируемое окно с рамкой и клиентской областью, а за ним сразу же автоматически диалоговое окно с кнопками. При этом, главное окно остается на экране, но теряет активность. Нажатие кнопок диалогового окна приводит к его закрытию. Примерный вид интерфейса показан ниже* (рисунок 81).

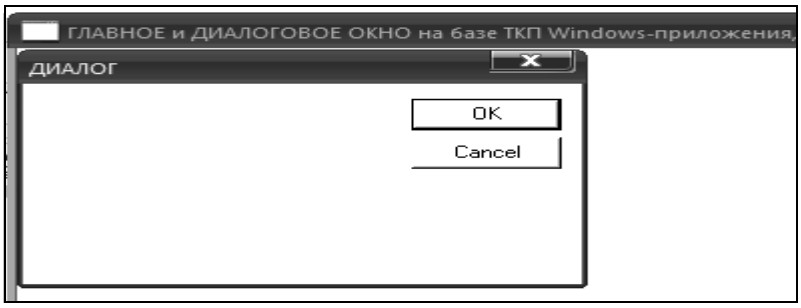


Рисунок 81 - Оконный интерфейс

Для этого: изучить теоретический материал и воспроизвести пример. Модифицировать приложение - добавить вывод подтверждения нажатия кнопок окна, разрушать диалоговое окно только при нажатии кнопки Cancel, при завершении работы с диалоговым окном разрушать и главное окно.

Порядок разработки (выполнения) задания.

1. Изучить результаты проектирования приложения (лекция 7, параграф "Диалоговое окно в составе главного окна"). Создать ТКП (на базе каркаса simple).

2. Описать глобальную переменную (например, HINSTANCE hInstance) для сохранения в ней дескриптора приложения, передаваемого из ОС в функцию WinMain, чтобы обеспечить доступ к дескриптору в любой точке приложения (и за пределами функции WinMain). Убедиться в работоспособности приложения, выполнив его.

3. Создать ресурсный файл и подключить его к проекту:

- создать файл описания ресурсов (или просто файл ресурсов *.rc), для чего выполнить ГМ-Project-Add To Project-New-Files, выбрать тип – Resource Script, а в качестве имени файла задать, например, <ИмяПриложения> (если ИмяПриложения – main, то соответственно после создания файла ресурсов *.rc в папке проекта появятся файлы с названиями main.rc, resource.h);

- подключить файл ресурсов к приложению посредством команды #include "resource.h"; - выполнить приложение (диалоговое окно не появится!).

4. Создать ресурс - диалоговое окно:

- добавить ресурс в ресурсный файл приложения, используя редактор ресурсов (при необходимости настроить редактор – командой главного меню Tools-Customize вызвать окно Customize, где на вкладке ToolBars включить режим Controls, вызывающий вывод меню элементов управления). Для этого вызвать его командой ГМ-Insert-Resource-Dialog. Добавить необходимые элементы управления в окно;

- при необходимости настроить их свойства (здесь три элемента – само окно и две кнопки). Для этого выделить элемент щелчком, а правой клавишей вызвать контекстное меню, в нем режим Properties и сделать необходимые установки;

- каждый элемент имеет свой идентификатор. Чтобы его увидеть, выделить элемент щелчком, а правой клавишей вызвать контекстное меню и в нем режим Properties.

Например, в свойствах окна определите его дескриптор - идентификатор ID (это может быть, например, ID = IDD_DIALOG1, хотя сам дескриптор можно поменять для повышения читаемости программы). Выписать дескрипторы всех элементов управления;

- выполнить приложение (диалоговое окно не появится!).

5. Описать функцию диалогового окна - программу обработчик (имя выбирается произвольно!) его сообщений с прототипом:

```
LRESULT CALLBACK   ИмяОбработчика (HWND, UINT, WPARAM, LPARAM);

LRESULT CALLBACK ИмяОбработчика ( HWND    hDlg, UINT   Message,
                                  WPARAM wParam,  LPARAM lParam)
{
     switch (Message)
     {
      case WM_INITDIALOG:
            return FALSE;
       case WM_COMMAND:
            switch (wParam)
            {
            case IDOK:
                  EndDialog(hDlg,TRUE);
                   break;
            case IDCANCEL:
                  EndDialog(hDlg,FALSE);
                   break;
             default:
                   return FALSE;
            }
            break;
       default:
            return FALSE;
      }
       return TRUE;
};
```

- выполнить приложение (диалоговое окно не появится!).

6. Инициализировать окно и вывести его на экран функцией, вставленной (на самом деле не корректно!) в пользовательскую секцию case WM_PAINT функции главного окна приложения, например, как DialogBox (hInstance, (LPCTSTR) IDD_DIALOG1, hWnd, (DLGPROC) TO_PROCESS_DIALOG_BOX));

- выполнить приложение - появится диалоговое окно.

Проверить, что при событиях, ведущих к сообщению WM_PAINT, диалоговые окна выводятся снова – иногда многократно. Диалоговое окно можно сбросить любой кнопкой, но по тем же причинам он может появиться снова. Поэтому изменить место создания диалогового окна (место вставки DialogBox). Вариант: а) после ShowWindow; б) перенести в секцию case WM_LBUTTONDOWN. И проверить, что при событиях, ведущих к сообщению WM_PAINT, диалоговые окна не выводятся многократно.

Задание 9. Использование диалогового окна в качестве главного. *Создать приложение, при запуске которого в качестве главного выводится диалоговое окно. Нажатие*

его кнопок приводит к выводу соответствующих сообщений, а нажатие Cancel - к его закрытию.

Порядок разработки (выполнения) задания.

1. Изучить результаты проектирования приложения (лекция 7, параграф “Диалоговое окно в качестве главного окна”). Создать приложение на базе ТКП.

2. Создать ресурс – диалоговое окно (например, с ID - IDD_DIALOG1) и подключить его к приложению командой #include "resource.h".

3. Внести изменения в текст приложения согласно листингу, например: а) описать пустой обработчик сообщений диалогового окна и его прототип

```
LRESULT CALLBACK   ИмяОбработчика (HWND hDlg, UINT Message,
                                          WPARAM wParam,  LPARAM lParam)
{
      return TRUE;
}   ;
```

б) в главной функции WinMain вместо действий по описанию, регистрации, созданию и визуализации классического главного окна (с рамкой) инициализировать и визуализировать диалоговое окно. Для этого связать текущее приложение (например, HINSTANCE hInst), ранее созданный ресурс - диалоговое окно (например, IDD_DIALOG1) и его обработчик командой DialogBox, например

```
DialogBox(hInst, (LPCTSTR) IDD_DIALOG1, NULL, (DLGPROC) ИмяОбработчика ) ;
```

в) описать обработчик

```
LRESULT CALLBACK ИмяОбработчика ( HWND    hDlg, UINT   Message,
                                          WPARAM wParam,  LPARAM lParam)
{
     switch (Message)
     {
      case WM_INITDIALOG:
            return FALSE;
      case WM_RBUTTONDOWN:
            < ФрагментПользователя >
            break;
       case WM_COMMAND:
            switch (wParam)
            {
            case IDOK:
                  < ФрагментПользователя >
                   break;
            case IDCANCEL:
                  < ФрагментПользователя >
                   break;
             default:
                   return FALSE;
             }
            break;
       default:
```

```
        return FALSE;
    }
    return TRUE;
```

ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОГО ВЫПОЛНЕНИЯ

1. Реализовать на базе ТКП приложение с диалоговым окном в качестве главного окна. Его функция один раз создать окно с клиентской областью и передать ему управление. При этом диалоговое окно может остаться на экране или может завершаться при запуске второго окна. В любом случае оно далее не оказывает никаких действий и завершается при закрытии второго окна – при завершении работы с приложением. Интерфейс показан ниже (рисунок 82).

Пояснения к выполнению. Конкретный экземпляр окна с рамкой создается функцией hWnd = CreateWindow(…), которая сообщает его дескриптор для дальнейшего использования в разных функциях. Само окно после визуализации активно, но если его закрыть, то конкретный экземпляр окна теряется как и его дескриптор. Поэтому для повторной визуализации окна того же стиля необходимо заново создать его экземпляр функцией CreateWindow и получить новый дескриптор. Соответственно в WinMain() создать стиль второго окна, создать его экземпляр, создать и активизировать диалоговое окно. Обрабатываемое сообщение диалогового окна: сообщение WM_COMMAND: IDOK по нажатии кнопки ОК. Действие – визуализация окна с рамкой, созданного в главной функции. Соответственно в обработчике диалогового окна по ОК визуализировать второе окно (команды ShowWindow(*ДескрипторОкнаСРамкой*, *ПараметрВизуализации*) и UpdateWindow (*ДескрипторОкнаСРамкой*), а секцию закрытия не использовать. Обрабатываемые сообщения второго окна: WM_RBUTTONDOWN по нажатии левой кнопки мыши в пределах окна. Действие – вывод информационного сообщения о событии; WM_DESTROY по закрытию окна. Действие – посылка сообщения на закрытие приложения PostQuitMessage(0). Соответственно в обработчике второго окна реагировать на события, производить пользовательскую обработку и закрытие приложения при свертывании окна.

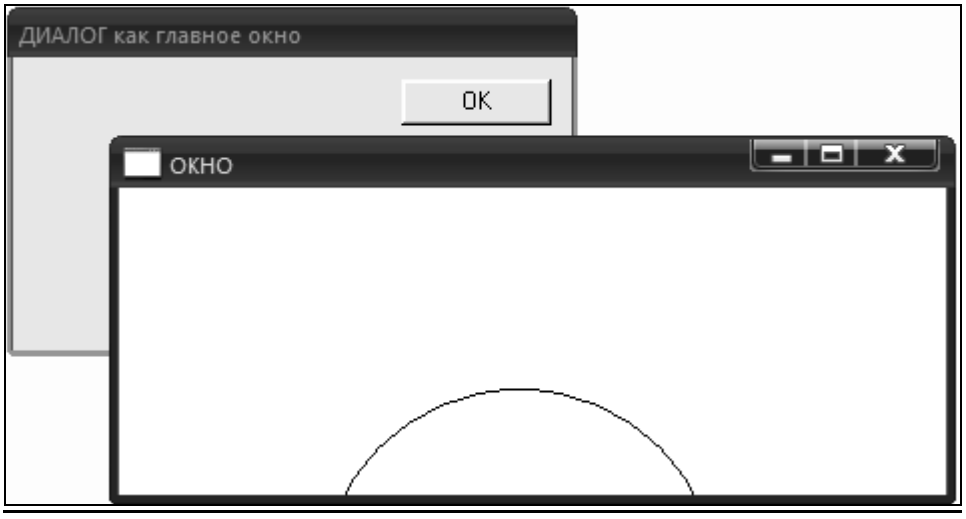


Рисунок 82 - Фрагмент интерфейса

2. Модифицировать предыдущее приложение с гашением диалогового окна после визуализации второго окна. Для этого сохранять дескриптор диалогового окна в глобальной переменной для последующего использования в обработчике окна с рамкой, например, HWND hDlgGlobal; LRESULT CALLBACK ОбработчикДиалоговогоОкна (…) { hDlgGlobal=hDlg; …….. } . Для гашения диалога в обработчике окна с рамкой об-

рабатывать сообщение, например, получение фокуса WM_SETFOCUS - EndDialog(hDlgGlobal,TRUE).

Рисунок 83 - Фрагмент интерфейса

3. Модифицировать приложение п.1 с многократным запуском окон одного стиля с рамкой по кнопке ОК. Интерфейс показан - (рисунок 83).

4. Модифицировать предыдущее приложение с гашением диалогового окна после визуализации второго окна.

5. Создать приложение с диалоговым окном в качестве главного с системной кнопкой закрытия и с единственной пользовательской кнопкой ОК, обеспечивающей многократный запуск окна с рамкой с системными кнопками при сохранении первого. Завершение приложения производить при закрытии любого из окон (родительского или дочерних).

6. Создать приложение с диалоговым окном в качестве главного с системной кнопкой закрытия и с пользовательскими кнопками – ОК и CANCEL, а также с реакцией на нажатие правой клавиши мыши. Кнопка ОК обеспечивает многократный запуск окна с рамкой с системными кнопками при сохранении первого. Завершение приложения по CANCEL, закрытие любого из дочерних окон – его системной кнопкой. Интерфейс показан - (рисунок 84).

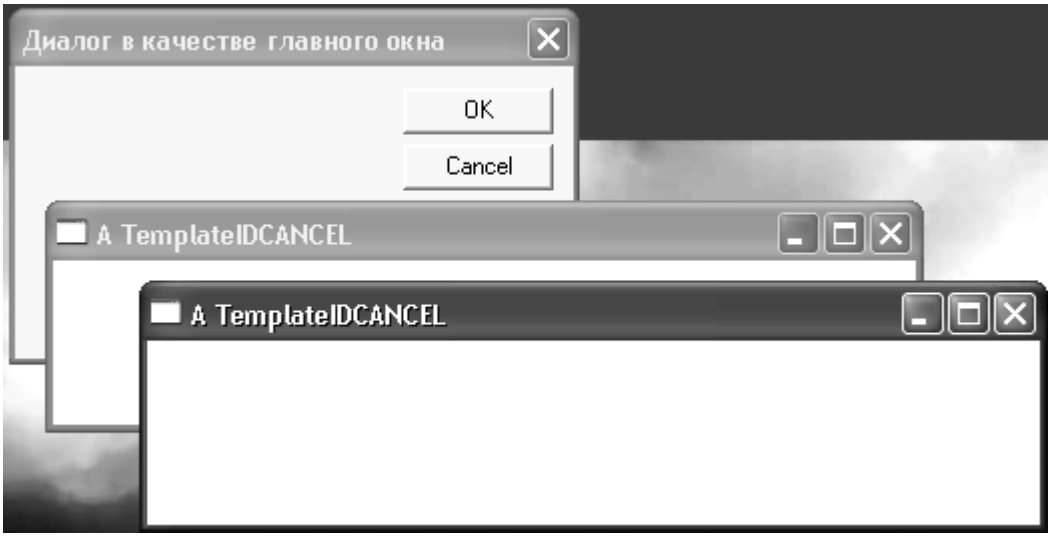


Рисунок 84 - Фрагмент интерфейса 3

7. Создать приложение с диалоговым окном в качестве главного с системной кнопкой закрытия и с пользовательскими кнопками – ОК и CANCEL, а также с реакцией на нажатие правой клавиши мыши. Кнопка ОК обеспечивает многократный запуск окна с рамкой с системными кнопками при сохранении первого. Завершение приложения по CANCEL, закрытие любого из дочерних окон – его системной кнопкой.

Элемент управления окно редактирования

Задание 10. Изучить теоретический материал по использованию ЭУ (лекция 7, Графические ресурсы).

Задание 11. Диалоговое окно с ЭУ редактирования. *Создать приложение, при запуске которого в качестве главного должно выводиться диалоговое окно с ЭУ редактирования, подсказками, кнопками), предназначенное для задания исходных данных и их эхо-вывода. По кнопке Ввести производится ввод из поля ввода и эхо-вывод введенного значения.* Вид диалогового окна приведен ниже (рисунок 85). Окно включает ЭУ редактирования (Edit Box), статичные окна (Static Text), кнопку Ввести.

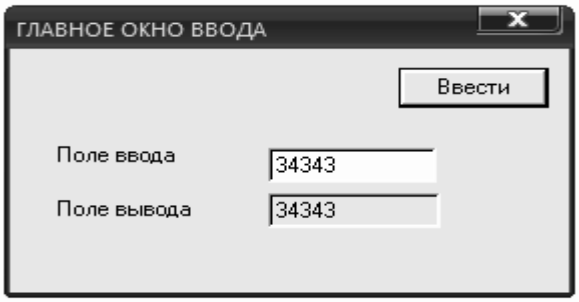


Рисунок 85 - Фрагмент интерфейса

Порядок разработки (выполнения) задания.

1. Изучить результаты проектирования приложения (лекция 7, параграф “Диалоговое окно в качестве главного окна – Диалоговое окно с окном редактирования). Создать приложение на базе ТКП.

2. Создать ресурс – диалоговое окно (например, с ID = IDD_DIALOG1), с окошками редактирования (например, с IDC_EDIT1, IDC_EDIT2 для ввода и вывода соответственно), со статическими окнами (например, с IDC_STATIC1, IDC_STATIC2 соответственно с названиями Caption - “Поле ввода”, “Поле вывода”), с кнопкой (например, с ID_Enter). Настроить свойства элементов управления. Например, для поля эхо-вывода с IDC_EDIT2 установить в окне его свойств (Edit Properties) способ выравнивания текста – к левому полю, режим работы – только чтение, автоматическая горизонтальная прокрутка. Проверить работу окна командой - главное меню, Layout, Test. Подключить его к приложению командой #include "resource.h".

3. Внести изменения в текст приложения: а) описать пустой обработчик сообщений диалогового окна и его прототип; б) в главной функции WinMain вместо действий по визуализации окна с рамкой визуализировать диалоговое окно командой DialogBox; в) описать обработчик. В секции case ID_Enter описать строковую переменную char InputString[256], ввести данные из окна ввода в строку командой GetDlgItemText(hDlg,IDC_EDIT1, InputString,15), вывести данные из строки в окно вывода командой SetDlgItemText(hDlg,IDC_EDIT2, InputString). В секции IDCANCEL выполнить EndDialog(hDlg,TRUE), PostQuitMessage(0).

4. Компилировать и наблюдать работу окна.

КОНТРОЛЬНЫЕ ЗАДАНИЯ

1. Разработать приложение на базе каркаса Hello для табуляции функции и вывода результатов в клиентскую область (КО) главного окна (ГО). 2. Разработать приложение с интерфейсом из двух окон – главного окна и диалогового (ДО), запускаемого щелчком правой клавиши мыши. Состав ДО: кнопка Отказ для его разрушения и возврата в ГО. Для этого прототипировать приложение, спроектировать и описать: - интерфейсные формы (окна), - диаграммы пользовательских классов (при использовании классов), - диаграмму состояний интерфейсных форм. 3. Разработать приложение (на базе предыдущего) с интерфейсом из двух окон – ГО и ДО, запускаемого щелчком правой клавиши мыши. Состав ДО: ЭУ для ввода массива. При завершении ввода выводить массив в КО главного окна. Все действия с данными (массивом чисел) описать, используя свой - пользовательский класс (классы). Для этого прототипировать приложение - спроектировать и описать: - интерфейсные формы, - диаграмму классов, - диаграммы состояний интерфейсных форм.

СПРАВОЧНЫЕ ДАННЫЕ. При выводе данных в клиентской области окна требуется вычислять координаты начальной позиции для вывода первого символа каждой строки, а при переходе на новую строку окна вычислять новое значение координаты y. Для определения новой координаты X, после вывода текущей строки, надо использовать метод MFC CDC::GetTextExtent() вычисления ее реальной длины в логических единицах

```
CSize CDC::GetTextExtent( LPCTSTR lpszString, int nCount ) const;
CSize CDC::GetTextExtent( const CString &str ) const;   .
```

Для строки lpszString из nCount = strlen(lpszString) символов метод возвращает ее длину в виде объекта класса CSize, унаследованного от структуры SIZE typedef struct tagSIZE { int cx; int cy; } SIZE .

В API используется аналогичная функция GetTextExtentPoint(HDC hdc, LPCTSTR lpszString, int nCount, LPSIZE lpSize) , где lpSize – указатель на результат - структуру типа SIZE. Для перехода на новую строку - вычисления нового значения координаты Y надо использовать данные из структуры TEXTMETRIC, которая хранит атрибуты (метрики) текущего шрифта, связанного с КУ. В том числе, поле tmHeight задает полную высоту символов используемого шрифта, tmExternalLeading определяет расстояние (интервал) между строками. Для доступа к структуре использовать метод MFC CDC::GetTextMetrics( ) либо функцию API - BOOL GetTextMetrics(HDC hdc, LPTEXTMETRIC lptm), где lptm – указатель на результат - структуру TEXTMETRIC.

ЛАБОРАТОРНАЯ РАБОТА № 3 “Создание интерфейса оконного windows-приложения (окна, меню, элементы управления). Автоматический каркас приложения”

ЦЕЛЬ РАБОТЫ. 1. Изучить использование типовых ЭУ (кнопок, окон редактирования, списков). 2. Изучить управление меню, технологию создания интерфейсов с использованием меню и диалоговых окон. 3. Изучить типовые диалоговые окна. 4. Изучить создание приложений на базе каркаса Hello.

СОСТАВ ОТЧЕТА. 1. По заданиям 2, 7 - диаграммы состояний. 2. По заданию 8: - формы интерфейса; - диаграмма состояний; - описание обработчиков; - диаграмма компонентов; - листинг приложения. 3. По заданию 9: - диаграмма состояний. 4. По заданию 11: - диаграмма состояний; - описание обработчиков; - листинг приложения.

ЗАДАНИЯ ДЛЯ ВЫПОЛНЕНИЯ.

Задание 1*. Выполнить контрольные задания.

Задание 2. Разработать приложение с диалоговым окном в качестве главного окна (на базе ТКП) с единственной кнопкой ОК и без системных кнопок. Его функция - создать окно с клиентской областью и передать ему управление. При этом диалоговое окно должно остаться на экране. Окно типа главное воспринимает сообщение нажатия левой кнопки мыши (действие – вывод информационного сообщения о событии; закрытие приложения). Приложение также может быть закрыто и системной кнопкой.

Модификации: а) блокировка кнопки ОК ДО при запуске ГО; б) попеременное нажатие кнопки ОК должно приводить к визуализации-девизуализации ГО.

ПОЯСНЕНИЯ. Конкретный экземпляр окна с рамкой создается функцией hWnd = CreateWindow(…), которая возвращает его дескриптор. Само окно после визуализации активно, но если его закрыть, то конкретный экземпляр окна теряется как и его дескриптор. Поэтому для повторной визуализации окна того же стиля необходимо заново создать его экземпляр функцией CreateWindow и получить новый дескриптор. Соответственно в WinMain( ) следует создать стиль второго окна, создать его экземпляр, создать и активизировать диалоговое окно.

Обрабатываемое сообщение диалогового окна - WM_COMMAND: IDOK, посылаемое по нажатии кнопки ОК. Действие – визуализация окна с рамкой, созданного в главной функции. Соответственно в обработчике диалогового окна для этого сообщения надо визуализировать второе окно - команды ShowWindow(*ДескрипторОкнаСРамкой*, *ПараметрВизуализации*) и UpdateWindow (*ДескрипторОкнаСРамкой*), а секцию закрытия следует не использовать и блокировать обработку повторного нажатия кнопки ОК.

Обрабатываемые сообщения второго окна: 1. Сообщение WM_RBUTTONDOWN. Действие – вывод информационного сообщения о событии. 2. Сообщение WM_DESTROY по закрытию окна. Действие – посылка сообщения на закрытие приложения PostQuitMessage(0). Соответственно в обработчике второго окна надо реагировать на эти события.

Задание 3. Разработать приложение с главным окном. Его функция – запуск при нажатии левой кнопки мыши диалогового окна, содержащего кнопку завершения работы приложения. При запуске ДО разрушать ГО без завершения работы приложения. Собственное разрушение ГО должно быть заблокировано.

Задание 4. Изучить теоретический материал по созданию и использованию пользовательского меню в составе главного окна (лекция 7, параграф “Использование ресурса меню”).

Задание 5. Воспроизвести ЗАДАНИЕ (лекция 7, параграф “Использование ресурса меню” - демонстрационный пример). *Разработать приложение на базе ТКП с окном с рамкой в качестве главного, содержащее простейшее пользовательское меню.* Примерный вид интерфейса показан ниже (рисунок 86). Здесь два пункта типа POPUP – ВВОД, ВЫВОД, четыре пункта типа MENUITEM – Строка 1, Строка 2 (подпункты пункта ВВОД), Строка 1, Строка 2 (подпункты пункта ВЫВОД).

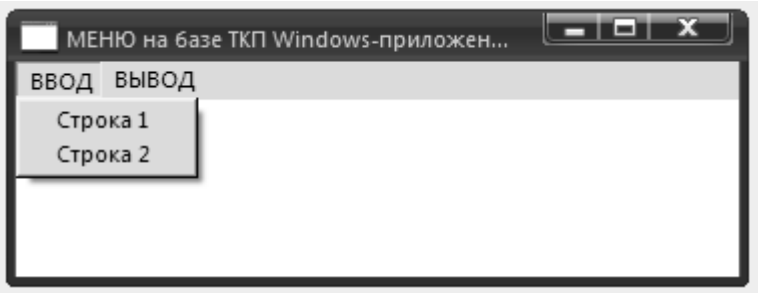


Рисунок 86 - Фрагмент интерфейса с меню

Порядок выполнения задания.

1. Создать ТКП
2. Создать ресурсный файл и подключить его к проекту командой #include "resource.h"; выполнить приложение (меню не появится!).
3. Спроектировать и описать ресурс:
- разработать вид и состав меню;
- добавить новый ресурс-меню в ресурсный файл приложения командой главного меню Insert-Resource-Menu;
- создать меню в редакторе ресурсов, указывая пункты меню и описывая их свойства. В свойствах меню определить его дескриптор, идентификатор ID (пусть, например, ID = IDR_MENU1); - выполнить приложение (меню не появится!).
4. Присоединить меню (его имя определено как IDR_MENU1) к приложению: - в функции WinMain() при описании стиля окна в переменной WNDCLASS wcApp определить ссылку на меню, используя поле

```
wcApp.lpszMenuName   = (LPTSTR) IDR_MENU1;
```

- выполнить приложение (меню визуализируется, но не реагирует на выбор пунктов).
5. Настроить обработку событий выбора пунктов меню, для чего внести изменения в обработчик, добавив к обработке событий WM_PAINT, WM_DESTROY фрагмент реакции на выбор пунктов меню, например

```
LRESULT CALLBACK WndProc (HWND hWnd,  UINT messg,
                                    WPARAM wParam,  LPARAM lParam)
{
    HDC hdc;
    PAINTSTRUCT ps;
    switch (Message)
    {
    case WM_PAINT:
            ……
            break;
    case WM_DESTROY:
            PostQuitMessage(0);
            break;
    case WM_COMMAND:
            switch (wParam)
            {
```

```
        case IDM_inString1:
                ……
                break;
        case IDM_inString2:
                ……
                break;
        case IDM_outString3:
                ……
                break;
        case IDM_outString4:
                ……
                break;

        default:
                return DefWindowProc(hWnd, messg, wParam, lParam);
        }
        break;
        default:
                return (DefWindowProc(hWnd, messg, wParam, lParam));
}
return 0;   }.
```

Здесь в качестве реакции на выбор пунктов меню выводить подтверждение сделанного выбора командой MessageBox. Выполнить приложение – убедиться в корректности работы меню.

Задание 6. Внести изменения в меню (подпункт “Строка2” пункта ”ВВОД” переименовать в “Выход” и генерировать сообщение для завершения работы приложения; - по выбору каждого конечного пункта меню выводить комментирующее сообщение; - подпункт “Строка1” пункта ”ВЫВОД” сделать типа POPUP и далее добавить список подпунктов “Дополнительный1”, “ Дополнительный2”).

Задание 7. Использование диалогового окна со списком и окошком редактирования в качестве главного. *Разработать приложение с диалоговым окном в роли главного. Цель приложения: управление списком строк – фамилий (просмотр, выбор, добавление, удаление, редактирование, расположение в алфавитном порядке). При запуске приложения в качестве главного окна выводится диалоговое, содержащее пустой список, кнопки управления списком и кнопку Завершить (Cancel).* Примерный вид окна представлен на рисунке 87. Состав ресурсов: - список, окно редактирования, кнопки, рамки – ЭУ типа Group Box и т.д.

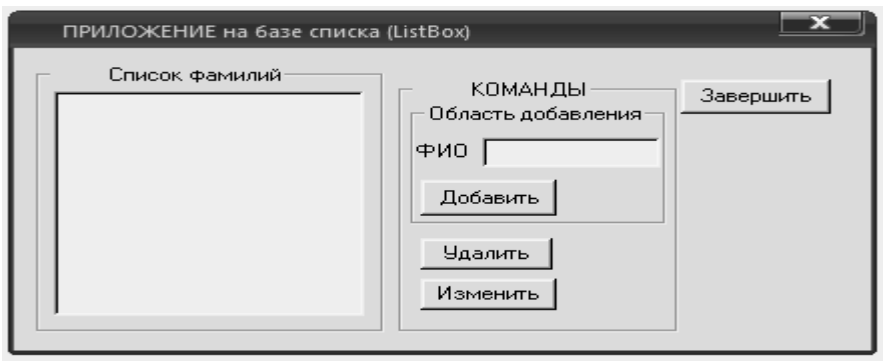


Рисунок 87 - Интерфейс приложения со списком

Порядок разработки (выполнения) задания.

1. Изучить результаты проектирования приложения (лекция 7, параграф “Диалоговое окно в качестве главного окна – диалоговое окно со списком и окном редактирования). Создать приложение на базе ТКП.

2. Создать и подключить ресурс – диалоговое окно.

3. Описать обработку сообщений в соответствии с их спецификациями (см. выше): - инициализация окна; - нажатие кнопки ЗАВЕРШИТЬ; - нажатие кнопки ДОБАВИТЬ; - нажатие кнопки УДАЛИТЬ; - нажатие кнопки ИЗМЕНИТЬ; - выбор строки в списке фамилий двойным щелчком; - выбор строки в списке фамилий щелчком.

Задание 8. Модифицировать приложение для работы со списком строк (задание 7), обеспечив константную инициализацию списка заранее заданными строками (фамилиями).

Задание 9. Разработать приложение на базе ТКП для многократного ввода-вывода строк и фиксации числа введенных строк. *Для этого Создать интерфейс на основе окна с рамкой и меню. Через меню вызывается диалоговое окно для ввода и окно сообщения для вывода строки. В клиентскую область главного окна выводится число введенных строк. Вид интерфейса показан ниже. Главное окно содержит меню с двумя пунктами. По пункту ВВОД выводится диалоговое окно ввода строки. По пункту ВЫВОД выводится строка и количество строк.*

Примерный вид интерфейса показан ниже (рисунок 88). Главное окно содержит пользовательское меню с двумя пунктами ВВОД и ВЫВОД. По пункту ВВОД выводится диалоговое окно (с окошком редактирования и кнопкой ВВЕСТИ), предназначенное для задания вводимой строки. По пункту ВЫВОД производится вывод ранее введенной строки. В нижней части окна выводится подсказка о количестве введенных строк. Пусть меню имеет идентификатор IDR_MENU1, пункты типа MENUITEM – ВВОД, ВЫВОД с идентификаторами IDM_In и IDM_Out, диалоговое окно имеет идентификатор IDD_DIALOG1, окно редактирования - IDC_EDIT1, кнопка ввода - ID_Enter.

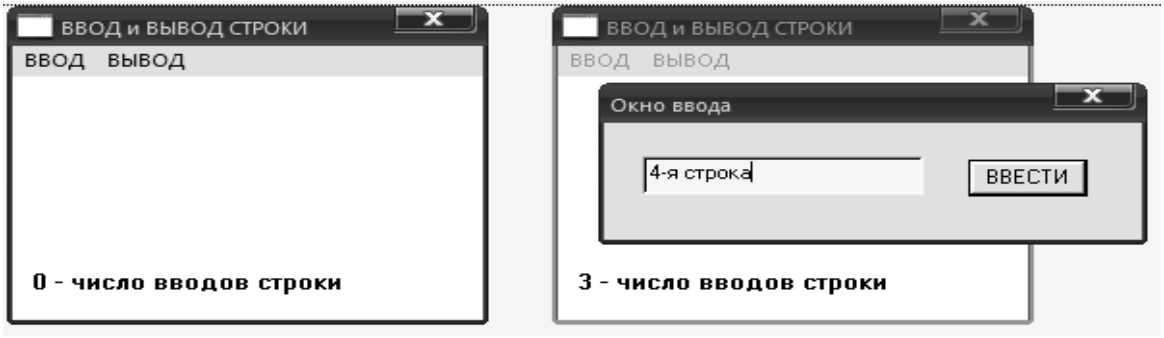


Рисунок 88 - Фрагмент интерфейса приложения с меню и окнами

Обрабатываемые сообщения. Это сообщения, адресуемые главному окну и обрабатываемые его обработчиком:

- сообщение WM_PAINT. Действие - вывести в клиентскую область окна новое значение количества вводов строки;
- сообщение WM_DESTROY. Действие - послать сообщение на завершение приложения;
- сообщение WM_COMMAND: IDM_In. Действие - инициализировать и вывести диалоговое окно ввода;
- сообщение WM_COMMAND: IDM_Out. Действие - вывести строку в окне сообщения.

Это сообщения, адресуемые диалоговому окну и обрабатываемые его обработчиком:

- сообщение WM_COMMAND: ID_Enter. Действия - увеличить количество вводов строки на единицу, ввести следующую строку из окна редактирования, закрыть диалоговое окно, послать сообщение на перерисовку главного окна.

ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОГО ВЫПОЛНЕНИЯ. 1. Модифицировать приложение, оформив все секции обработчика в виде системы функций. 2. Модифицировать приложение, реализуя вывод строки через отдельное окно вывода. 3. Модифицировать приложение, реализуя вывод-вывод строки в одном диалоговом окне. 4. Модифицировать приложение, реализуя вывод-вывод массива строк. Вывод строк производить либо в клиентской области окна, либо в ЭУ список диалогового окна. 5. Модифицировать приложение, сохраняя и загружая массив строк из заданного файла.

Задание 10. Изучить теоретический материал по использованию типовых диалоговых окон (лекция 7, параграф “Использование стандартных диалоговых окон”).

Создать приложение с окном с рамкой. По сообщению WM_ LBUTTONDOWN выводить окно типа “Сохранить как”. По сообщению WM_RBUTTONDOWN выводить окно типа “Открыть”.

Модифицировать приложение. Создать приложение с окном. По сообщению WM_RBUTTONDOWN визуализировать диалоговое окно типа “Открыть” с последующим открытием выбранного файла, по сообщению WM_LBUTTONDOWN визуализировать диалоговое окно типа “Сохранить как” с последующим сохранением информации в выбранном файле.

Примечание: для открытия файла используется функция CreateFile. Например, для открытия файла с именем szFileName для выполнения операций чтения

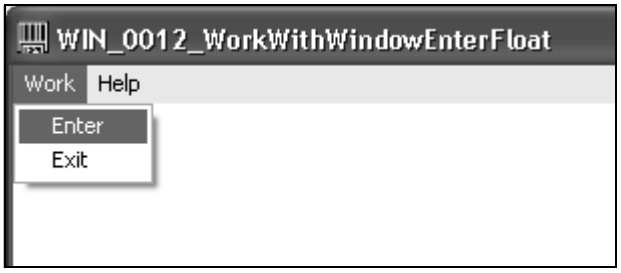
hFile = CreateFile(szFileName, GENERIC_READ, 0,NULL, OPEN_EXISTING,0,NULL) ,

а для открытия файла с именем szFileName для выполнения операций записи

hFile = CreateFile(szFileName, GENERIC_READ | GENERIC_WRITE, 0, NULL,
CREATE_ALWAYS, 0, NULL) .

Для чтения и записи данных используются функции ReadFile, WriteFile. Например, для чтения одной записи - строки из файла szFileName (дескриптор hFile) во внутреннюю строку char StrIn - ReadFile(hFile, StrIn, 9, &dwCount, NULL); а для записи строки в файл WriteFile(hFile, StrOut, sizeof(*StrOut), &dwCount, NULL); . Здесь DWORD dwCount.

Задание 11. Создать проект на базе шаблона Hello. Модифицировать меню приложения. По выбору подпункта Enter активизировать диалоговое окно для ввода числа. Введенное вещественное число возводить в квадрат, а полученный результат выводить с помощью окна сообщений. Фрагменты интерфейса приведены ниже - рисунок 89. Фрагменты интерфейса приложения на базе ТКП Hello.



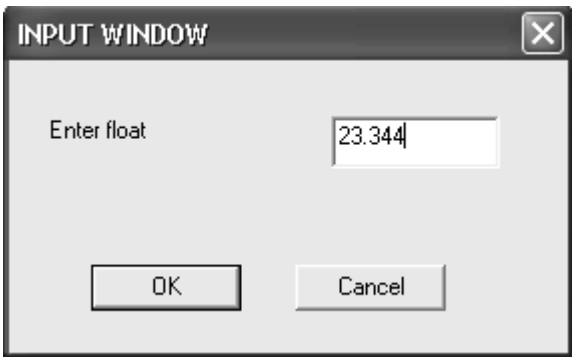


Рисунок 89 - Фрагменты интерфейса приложения

КОНТРОЛЬНЫЕ ЗАДАНИЯ

1. Разработать приложение с интерфейсом из двух окон – главного окна и диалогового, запускаемого щелчком правой клавиши мыши. Состав диалогового окна: ЭУ список и другие ЭУ для его управления в соответствии с заданием. Обеспечить константную инициализацию списка (3-4 записи) и выполнение команды Удалить для выделенной записи списка. Обеспечить выполнение команды Добавить для внесения в список новой записи. Обеспечить вывод текущего содержимого списка в клиентскую область главного окна при завершении работы с диалоговым. Для работы с данными (списком чисел) описать пользовательский класс (классы). Обеспечить сохранение и инициализацию списка из “системного” файла.

2. Разработать приложение с диалоговым окном в качестве главного окна и списком для работы с номерами телефонов.

3. Модифицировать приложение (задание 9 текущей работы). Оформить все секции обработчика в виде системы функций. Модифицировать приложение, реализуя вывод-вывод массива строк. Вывод строк производить либо в клиентской области окна, либо в ЭУ список диалогового окна. Модифицировать приложение, сохраняя и загружая массив строк из заданного файла.

4. Модифицировать приложение (задание 10 текущей работы). Реализовать запись в заданный файл и считывания из заданного файла константного массива строк (например, строк вида "Запись1", "Запись2" и т.д.). Модифицировать приложение для записи в заданный файл и считывания из заданного файла произвольного массива строк. 4.3. Модифицировать предыдущее приложение, добавив управление приложением через меню.

5. Разработать приложение с использованием меню и диалоговых окон. 5.1. Для ввода вещественных данных и вывода результатов их обработки. 5.2. Для ввода массива вещественных данных и вывода результатов их обработки.

6. Разработать приложение для табулирования заданных функций (параметры табулирования вводит пользователь; результаты выводятся как в табличном так и графическом виде в одном или разных окнах).

ЛАБОРАТОРНАЯ РАБОТА № 4 “Типовой каркас оконного mfc-приложения (ТКП). Обработка сообщений. Работа с клиентской областью окна”

ЦЕЛЬ РАБОТЫ. 1. Изучение типового каркаса MFC-приложения (ТКП). 2. Изучение организации обработки сообщений. 3. Организация вывода в клиентскую область главного окна, поддержка перерисовки. 4. Изучение элементов автоматизации разработки mfc-приложений.

СОДЕРЖАНИЕ ОТЧЕТА. 1. Описание ТКП (интерфейс, диаграммы классов и взаимодействия объектов). 2. Для приложений заданий 7, 8 - диаграммы классов и компоновки. 3. Описание ТКП, построенного “мастером” (интерфейс, состав файлов, классов, ресурсов, диаграммы классов, диаграммы взаимодействия объектов). 4. Описание приложений заданий 11, 13.

ЗАДАНИЯ ДЛЯ ВЫПОЛНЕНИЯ.

Задание 1. Изучить теоретический материал: - ознакомиться с общими сведениями об ОС, технологиях программирования, библиотеке MFC; - изучить основные типы сообщений, прототипы соответствующих обработчиков; - ознакомиться с методами управления контекстом устройства, перерисовкой; - ознакомиться со средой разработки; - ознакомиться с составом классов ТКП; - изучить технологию обработки сообщений; - изучить архитектуру ТКП, используемые классы и методы (лекция 8, параграфы “Особенности mfc-программирования в ОС Windows, Использование Visual Studio, Структура типового MFC-приложения”, лекция 9 “Обработка сообщений”).

Задание 2. Создать “пустое” приложение (ТКП) (лекция 8, параграфы “Использование Visual Studio - настройка Visual Studio”, “Структура типового MFC-приложения”). Это каркас приложения без обработки сообщений. Запустить. Изучить его свойства. Разработать диаграммы UML.

Задание 3. Модифицировать приложение, включив чувствительность к сообщению WM_LBUTTONDOWN (см. лекцию 9, параграф “Сообщения. Обработчики сообщений”). Для этого в классе CMainWin включить (разкомментировать) прототип функции-обработчика afx_msg void OnLButtonDown(UINT flags, CPoint loc); - в карте сообщений BEGIN_MESSAGE_MAP() включить (разкомментировать) чувствительность приложения к нажатию клавиши “мыши” ON_WM_LBUTTONDOWN();

- описать (разкомментировать) соответствующую функцию-обработчик; - выполнить приложение, исправить ошибки; - добавить в тело обработчика вывод сообщения о его запуске с помощью метода MessageBox (см. лекцию 9, параграф “Организация ввода-вывода – методы ввода-вывода”); - добавить в тело обработчика вывод текста “Работает ТКП-MFC” в главном окне с помощью метода TextOut. Для этого получить контекст устройства – создать объект класса CClientDC (лекция 9, параграф “Организация ввода-вывода – контекст устройства”); - запустить приложение, выполнить манипуляции с окном (перемещение, изменение размеров, перекрывание другим приложением) и проанализировать эффект перерисовки.

Задание 4. Создать приложение (на базе ТКП), аналогичное созданному в предыдущем пункте, но включив чувствительность к сообщению WM_PAINT (лекция 9, параграф “Сообщения. Обработчики сообщений”). Для этого в класс CMainWin включить прототип функции-обработчика afx_msg void OnPaint(); - в карте сообщений BEGIN_MESSAGE_MAP( ) включить чувствительность приложения к сообщению перерисовки ON_WM_PAINT();

- описать соответствующую функцию-обработчик; - выполнить приложение, исправить ошибки; - добавить в тело обработчика вывод сообщения о его запуске с помощью метода MessageBox; - добавить в тело обработчика вывод текста “Работает ТКП-MFC” в главном окне с помощью метода TextOut. Для этого получить контекст устройства – создать объект класса CPaintDC; - запустить приложение, выполнить манипуляции с окном (перемещение, изменение размеров, перекрывание другим приложением) и анализиро-

вать эффект перерисовки; - добавить табуляцию любой функции с выводом результатов в табличном виде (аргумент-функция).

Задание 5. Создать приложение для вывода координат курсора “мыши”, фиксируемых по щелчку ее левой клавиши. Для этого: - создать ТКП и включить чувствительность к сообщению WM_LBUTTONDOWN; - обеспечить вывод координат курсора “мыши”, фиксируемых по щелчку ее левой клавиши, в обработчике сообщения WM_LBUTTONDOWN; - запустить приложение, выполнить манипуляции с окном и анализировать эффект перерисовки.

Задание 6. Модифицировать приложение, решив проблему перерисовки - ввод координат производить по сообщению WM_LBUTTONDOWN, вывод по WM_PAINT. Для этого: - включить чувствительность к сообщению WM_PAINT; - модифицировать обработчик сообщения WM_LBUTTONDOWN, осуществляя в нем получение и запоминание координат курсора момента нажатия левой клавиши “мыши”; обеспечить вывод координат курсора в обработчике сообщения WM_PAINT. Запустить приложение, выполнить манипуляции с окном и анализировать эффект перерисовки; - для вызова перерисовки окна после каждого щелчка левой клавиши “мыши” в обработчике сообщений “мыши” использовать метод InvalidateRect (лекция 9, параграф “Организация ввода-вывода - перерисовка”); - запустить приложение, выполнить манипуляции с окном и анализировать эффект перерисовки.

Задание 7. Модифицировать приложение для вывода координат в той точке экрана, где был зафиксирован щелчок левой клавиши “мыши”.

Задание 8. Создать приложение для ввода символа с клавиатуры (сообщение WM_CHAR) и эхо-вывода либо в левый верхний угол главного окна, либо в позицию, задаваемую нажатием левой клавиши мыши. Разработать диаграммы UML.

Примечание. Ввод и запоминание символа выполнять в обработчике сообщения WM_CHAR, ввод и запоминание координат выполнять в обработчике сообщения WM_LBUTTONDOWN, вывод выполнять в обработчике сообщения WM_PAINT. Вызывать перерисовку при вводе нового символа или при изменении координат вывода методом InvalidateRect(NULL).

Задание 9. Создать “пустое” приложение (ТКП) с разными вариантами стиля главного окна (лекция 8, параграф “Особенности mfc-программирования в ОС Windows - класс CFrameWnd”). В том числе, создать главное окно: - без возможности скроллинга в клиентской области; - со скроллингом в клиентской области окна; - окно без системных кнопок; - окно заданного размера и местоположения (например, с координатами левого верхнего угла 10, 10 и правого нижнего - 200, 200) без системных кнопок (для этого использовать структуру типа RECT, а затем класс CRect.. Пример задания параметров клиентской области приведен далее) RECT TheRect = {10, 10, 200, 200} или RECT TheRect; TheRect..top = 10; TheRect..left = 10; TheRect..bottom = TheRect.right = 200..

Задание 10. Элементы автоматизации.

Задание 10.1. Создание каркаса MFC-приложения (ТКП) с помощью мастера. Для этого запустите мастер MFC (например, AppWizard (exe)). В поле Project name введите название проекта приложения и нажмите кнопку ОК. В окне мастера выберите - однодокументное приложение (Single Document Interface - SDI, флажок - Single document) и откажитесь от поддержки архитектуры Документ-Вид (сбросьте флажок – Document/View architecture support). Остальные опции сохраните со значениями по умолчанию. В версиях студии с поддержкой NET выберите – приложение MFC с одним документом и без поддержки архитектуры Document/View. Запустите приложение. Изучите его состав и функциональные возможности.

Задание 10.2. Добавление обработчиков. Во всех каркасах MFC поддерживаются элементы автоматизации. Так включение макросов в карты сообщений и добавление

соответствующих обработчиков сообщений может быть выполнено, например, мастером ClassWizard. Пусть надо добавить обработчик события - нажатие левой клавиши мыши. Для этого надо: - открыть ClassWizard, нажав Ctrl-W; - перейти на вкладку Message Maps; - выбрать в списке Class Name – класс соответствующего окна; - выбрать в списке Object IDs аналогичный пункт; - в списке Messages выбрать нужное сообщение (здесь WM_LBUTTONDOWN); - указать Add Function, а затем Edit Code и добавить необходимые действия в обработчик.

В версиях студии с поддержкой NET – в окне Классов выберите соответствующий класс (например, класс, поддерживающий окно – клиентскую область – CChildView). Для него в окне Свойств на вкладке Сообщения выберите нужное, добавьте обработчик (здесь выберите сообщение WM_LBUTTONDOWN и согласитесь с добавкой системного обработчика), внесите в обработчик необходимые действия. Далее используйте эти возможности при разработке приложений.

Задание 11. Создать приложение на базе ТКП для рисования плоских изображений отрезками прямых линий. Координаты точек вводить нажатием левой клавиши “мыши”. Введенные координаты сохранять в динамическом массиве (например, использовать vector). Выводить изображение, путем соединения прямыми линиями соседних точек, по нажатию правой клавиши “мыши”.

ПРИМЕЧАНИЕ. В начальном варианте реализовать только ввод, хранение и вывод координат. Для этого создать ТКП и обработчики сообщений. Для рисования точек, линий, геометрических фигур используются методы класса CDC, поэтому необходимо создать объект этого класса. Для хранения координаты точки можно использовать структуру POINT, класс CPoint. Для перемещения пера (без рисования линии) используется метод CDC::MoveTo(int x, int y) или CDC::MoveTo(POINT point). Для рисования прямой линии используется метод CDC::LineTo(int x, int y) или CDC::LineTo(POINT point).

Задание 12*. Модифицировать предыдущее приложение, использовав пользовательский класс ЛИНИИ для работы с координатами точек, линиями изображения.

Задание 13. Создать приложение на базе ТКП, которое выводит строки, например, как показано ниже (рисунок 90).

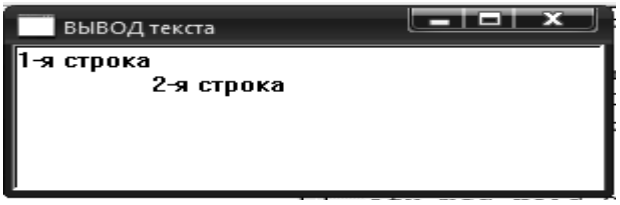


Рисунок 90 - Фрагмент интерфейса

ПРИМЕЧАНИЕ. При выводе данных в клиентской области экрана требуется вычислять координаты начальной позиции для вывода первого символа каждой строки, а при переходе на новую строку окна вычислять новое значение координаты y. Для определения новой координаты X, после вывода текущей строки, следует использовать метод CDC::GetTextExtent() вычисления ее реальной длины в логических единицах. Для перехода на новую строку окна и вычисления нового значения координаты Y надо использовать данные из структуры TEXTMETRIC, которая хранит атрибуты (метрики) текущего шрифта, связанного с КУ. В том числе, поле tmHeight задает полную высоту символов используемого шрифта, tmExternalLeading определяет расстояние (интервал) между строками. Для доступа к структуре использовать метод CDC::GetTextMetrics( ).

Задание 14*. Создать приложение, имитирующее работу упрощенного текстового редактора.

ЛАБОРАТОРНАЯ РАБОТА № 5. “Создание интерфейса оконного mfc -приложения (окна, диалоговые окна, меню, элементы управления)”

ЦЕЛЬ РАБОТЫ: 1. Ознакомиться с технологией работы с диалоговыми окнами и меню. 2. Изучить приемы использования меню, диалогового окна в составе главного окна и в качестве главного окна.3. Изучить приемы использования типовых элементов управления (ЭУ) диалогового окна.

СОДЕРЖАНИЕ ОТЧЕТА: - описание особенностей создания и управления приложением на базе ТКП с диалоговым окном (описание интерфейса, событий и сообщений, диаграммы используемых классов, диаграммы взаимодействия объектов); - описание особенностей создания и управления меню; - описание приложений (результатов выполнения самостоятельных заданий).

ЗАДАНИЯ ДЛЯ ВЫПОЛНЕНИЯ

Задание 1. Создать *MFC-приложение на базе ТКП с оконным интерфейсом из главного окна и диалогового окна* в составе главного *с двумя кнопками OK, CANCEL* со стандартными обработчиками сообщений. *При запуске приложения должно выводиться главное окно и на фоне главного окна приложения - диалоговое окно, которому и передается управление. Диалоговое окно функционирует в модальном режиме и после нажатия любой из кнопок OK или CANCEL исчезает, возвращая управление главному окну. Диалоговое окно появляется каждый раз при событии “перерисовки” главного окна, т.е. при попытке изменения размеров главного окна.*

Порядок разработки (выполнения) задания.

1. Изучить результаты проектирования приложения (лекция 10, параграф “Диалоговое окно в составе главного окна – пример-задание 1”).

2. Подготовить (реализовать) каркас MFC-приложения с файлом ресурсов.

2.1. Создать типовой каркас приложения. Убедиться в его работоспособности.

2.2. Создать в составе приложения файл описания ресурсов Resource Script (или просто файл ресурсов) с расширением - rc. Для этого выполнить команду ГМ-Project-AddToProject-New-Files, в качестве типа добавляемого файла указать – Resource Script, а в качестве имени файла задать любое имя, например, <ИмяПриложения>. Автоматически к проекту будет добавлен Resource Script файл с именем <ИмяПриложения>.rc>. Здесь ИмяПриложения – main.

2.3. Подключить файл описания ресурсов к приложению посредством команды #include "resource.h".

2.4. Выполнить приложение (диалоговое окно не появится!).

3. Создать новый ресурс - диалоговое окно.

3.1. Добавить ресурс к проекту приложения командой главного меню Insert-Resource-Dialog. Окну автоматически будет присвоен дескриптор (идентификационным номером), например ID - IDD_DIALOG1. Дескриптор можно посмотреть в свойствах окна и при желании его можно изменить. Для этого достаточно вызвать контекстное меню для шаблона окна, выполнить пункт properties и считать данные из окна свойств “Dialog Properties”. Кроме этого в поле редактирования должна появиться и панель инструментов (Controls).

В случае отсутствия панель инструментов можно активизировать через главное меню, выполнив пункты Tools-Customize. В появившемся окне Customize на вкладке Toolbars следует включить Controls.

3.2. Отредактировать внешний вид (облик) окна. Например, изменить размер и положение окна, размеры и расположение кнопок, перемещая их “мышью”. Настроить свойства (атрибуты) окна, используя вкладки окна свойств “Dialog Properties”. Например, задать стиль (Styles) окна как Popup и (Border) как окно с рамкой Dialog Frame (или Resizing!). Задать название заголовка окна Caption, например, “Образец диалогового окна”. Аналогично настроить свойства (атрибуты) кнопок (специализированных окон), используя вкладки окна свойств “Push Button Properties”.

Заметьте, что для системных кнопок ID установлены как IDOK, IDCANCEL. Это обеспечит автоматическую стандартную обработку событий их выбора – завершение работы с модальным диалоговым окном, стирание его с экрана. Проверьте работу окна – пункт главного меню Layout-Test.

3.3. Откомпилировать приложение и запустить на выполнение (диалоговое окно не появится!). Теперь можно просмотреть в текстовом редакторе (например, в Word) содержимое ресурсного файла (здесь main.rc), находящегося в папке проекта. Фрагменты содержимого приведены ниже.

```
//Microsoft Developer Studio generated resource script.
#include "resource.h"
 ……..
////////////////////////////////////////////////////////////////////////////
// Dialog
IDD_DIALOG1        DIALOG                    DISCARDABLE          0, 0, 186, 44
STYLE              DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION            "Образец диалогового окна"
FONT               8,  "MS Sans Serif"
BEGIN
    DEFPUSHBUTTON     "OK",         IDOK,          16, 13, 50, 14
    PUSHBUTTON        "Cancel",     IDCANCEL,      107, 13, 50, 14
END
////////////////////////////////////////////////////////////////////////////
……..
```

Ниже представлено мопутствующее содержимое файла resource.h. В нем можно получить числовые эквиваленты дескрипторов. Например, видно, что поименованному дескриптору (ID) диалогового окна IDD_DIALOG1 соответствует число 101. В дальнейшем ссылаться на соответствующий объект можно, используя любое представление ID.

```
//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by main.rc
//
#define IDD_DIALOG1                 101
// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE      103
#define _APS_NEXT_COMMAND_VALUE       40001
#define _APS_NEXT_CONTROL_VALUE       1000
#define _APS_NEXT_SYMED_VALUE         101
#endif
```

```
#endif
```

Состав ресурсов можно увидеть в соответствующем окне ResourceView системы Visual Studio.

4. Описать пользовательский класс для создания экземпляров пользовательских диалоговых окон. Здесь это класс MY_DIALOG, производный от CDialog и обслуживающий диалоговое окно IDD_DIALOG1:

```
class  MY_DIALOG: public CDialog
{
public:
     MY_DIALOG ( char  * DialogName, CWnd  *Owner ): CDialog ( DialogName, Owner)
                    { };
     //afx_msg прототипы обработчиков сообщений
     DECLARE_MESSAGE_MAP( )
};
```

Тогда каркас карты сообщений выглядит как

```
BEGIN_MESSAGE_MAP (MY_DIALOG, CDialog)
   //Макрокоманды указания типов обрабатываемых сообщений и их обработчиков
END_MESSAGE_MAP ()   .
```

В составе класса MY_DIALOG необходимо описать конструктор MY_DIALOG (char *DialogName, CWnd *Owner), где *DialogName* – дескриптор диалогового окна в строковом формате, *Owner* - указатель окна - владельца, родителя диалогового окна (например, текущее окно. Тогда в качестве Owner можно использовать указатель this). При необходимости описываются пользовательские обработчики сообщений (здесь не используются). Компилировать и выполнить (диалоговое окно не появится!).

5. Подключить диалоговое окно к приложению – оно будет запускаться сообщением WM_PAINT.

5.1. Включить в описание класса WINDOW главного окна обработчик сообщения WM_PAINT, т.е. описать функцию afx_msg void OnPaint() как член класса WINDOW:

```
class  WINDOW : public CFrameWnd
{
public:
     WINDOW( );
     afx_msg void OnPaint( );
     DECLARE_MESSAGE_MAP( )
};                    .
```

5.2. Включить в карте сообщений (очереди сообщений) главного окна (класса WINDOW) чувствительность к сообщениям WM_PAINT:

```
BEGIN_MESSAGE_MAP (WINDOW, CFrameWnd)
   ON_WM_PAINT( )
END_MESSAGE_MAP( )   .
```

5.3. Создать каркас функции-обработчика сообщения afx_msg void OnPaint()

```
afx_msg void WINDOW::OnPaint () {   CPaintDC dc ( this );
     // <ФРАГМЕНТ ПОЛЬЗОВАТЕЛЯ> };
```

5.4. Описать функцию-обработчик сообщения afx_msg void OnPaint(). Создать в ней экземпляр класса MY_DIALOG с именем TheDialog, инициализировать его параметрами (обликом) ресурса типа “диалоговое окно” с дескриптором ID = IDD_DIALOG1. Визуализировать и активизировать окно методом DoModal () :

```
afx_msg void WINDOW::OnPaint()
{
    CPaintDC  dc ( this );
    MY_DIALOG  TheDialog ( ( LPTSTR ) IDD_DIALOG1 , this );
    TheDialog.DoModal ( );
};    .
```

6. Откомпилировать и выполнить приложение. Тестировать работу приложения.

Задание 2. Создать MFC-приложение на базе *ТКП с оконным интерфейсом из главного и диалогового окна с двумя кнопками OK, CANCEL* и с переопределенными обработчиками сообщений. *При запуске приложения должно выводиться главное окно. При щелчке левой клавишей мыши в главном окне должно активизироваться диалоговое окно и выводиться на фоне главного окна приложения. Диалоговое окно функционирует в модальном режиме и после нажатия любой из кнопок (OK или CANCEL) исчезает, возвращая управление главному окну. При нажатии кнопки OK должно также выводиться сообщение, подтверждающее факт нажатия именно этой кнопки.*

Порядок разработки (выполнения) задания.

1. Изучить результаты проектирования приложения (лекция 10, параграф “Диалоговое окно в составе главного окна - переопределение обработчиков, пример-задание 2”).

2. Подготовить (реализовать) каркас MFC-приложения с файлом ресурсов.

2.1. Создать типовой каркас приложения. Выполнить и убедиться в его работоспособности.

2.2. Создать в составе приложения файл описания ресурсов Resource Script (или просто файл ресурсов) с расширением - rc. Для этого выполнить команду ГМ-Project-AddToProject-New-Files, в качестве типа добавляемого файла указать – Resource Script, а в качестве имени файла задать любое имя, например, <ИмяПриложения>. Автоматически к проекту будет добавлен Resource Script файл с именем <ИмяПриложения>.rc>. Здесь ИмяПриложения – main. Соответственно файловый (модульный) состав приложения представлен ниже и включает два файла.

2.3. Подключить файл описания ресурсов к приложению посредством команды #include "resource.h".

2.4. Выполнить приложение (диалоговое окно не появится!).

3. Создать новый ресурс - диалоговое окно.

3.1. Добавить ресурс к проекту приложения командой главного меню Insert-Resource-Dialog. Появится шаблон окна. Окну автоматически будет присвоен дескриптор (идентификационным номером), например, ID - IDD_DIALOG1.

3.2. Отредактировать внешний вид (облик) окна.

Например, изменить размер и положение окна, размеры и расположение кнопок, перемещая их “мышью”. Настроить свойства (атрибуты) окна, используя вкладки окна свойств “Dialog Properties”. Например, задать стиль (Styles) окна как Popup и (Border) как окно с рамкой Dialog Frame (или Resizing!). Задать название заголовка окна Caption.

Аналогично настроить свойства (атрибуты) кнопок (специализированных окон), используя вкладки окна свойств “Push Button Properties”. Проверьте работу окна – пункт главного меню Layout-Test.

3.3. Откомпилировать приложение и запустить на выполнение (диалоговое окно не появится!).

4. Описать пользовательский класс MY_DIALOG, обслуживающий диалоговое окно IDD_DIALOG1, как производный от библиотечного класса CDialog:

```
class  MY_DIALOG: public CDialog
{
public:
     MY_DIALOG ( char  * DialogName, CWnd  *Owner ): CDialog ( DialogName, Owner)
                    {   };
     afx_msg void OnButtonOK ( );
     DECLARE_MESSAGE_MAP( )
};
```

и карту сообщений

```
BEGIN_MESSAGE_MAP (MY_DIALOG, CDialog)
      ON_COMMAND(IDOK, OnButtonOK)
END_MESSAGE_MAP ( )   .
```

В составе класса MY_DIALOG необходимо описать конструктор MY_DIALOG (char *DialogName, CWnd *Owner), где DialogName – дескриптор диалогового окна в строковом формате, Owner - указатель окна - владельца, родителя диалогового окна (например, текущее окно. Тогда в качестве Owner можно использовать указатель this).

Переопределить обработчик

```
afx_msg void MY_DIALOG::OnButtonOK()
{
     MessageBox("OnOK","OnButtonOK");
     EndDialog(0);
};
```

Откомпилировать и запустить на выполнение (диалоговое окно не появится!).

5. Подключить диалоговое окно к приложению – оно будет запускаться сообщением ON_WM_LBUTTONDOWN.

5.1. Включить в описание класса WINDOW главного окна обработчик сообщения ON_WM_LBUTTONDOWN, т.е. описать функцию afx_msg void OnLButtonDown(UINT flags, CPoint loc) как член класса WINDOW:

```
class WINDOW : public CFrameWnd
{
public:
     WINDOW( );
     afx_msg void OnPaint();
     afx_msg void OnLButtonDown(UINT flags, CPoint loc);
     DECLARE_MESSAGE_MAP()
```

```
};
```

5.2. Включить в карте сообщений (очереди сообщений) главного окна (класса WINDOW) чувствительность к сообщениям типа WM_PAINT и WM_LBUTTONDOWN:

```
BEGIN_MESSAGE_MAP(WINDOW, CFrameWnd)
  ON_WM_PAINT()
  ON_WM_LBUTTONDOWN()
END_MESSAGE_MAP()  .
```

5.3. Создать каркас функции-обработчика сообщения afx_msg void OnPaint()

```
afx_msg void WINDOW::OnPaint()
{
     CPaintDC dc(this);
     // < ФРАГМЕНТ_ПОЛЬЗОВАТЕЛЯ >
};
```

5.4. Создать каркас функции-обработчика сообщения afx_msg void OnLButtonDown(UINT flags, CPoint loc)

```
afx_msg void WINDOW::OnLButtonDown(UINT flags, CPoint loc)
{
     CClientDC dc(this);
    // < ФРАГМЕНТ_ПОЛЬЗОВАТЕЛЯ >
};
```

5.5. Описать функцию-обработчик сообщения WM_LBUTTONDOWN - afx_msg void OnLButtonDown(UINT flags, CPoint loc). Создать в ней экземпляр класса MY_DIALOG с именем TheDialog, инициализировать его параметрами (обликом) ресурса типа “диалоговое окно” с дескриптором ID = IDD_DIALOG1. Визуализировать и активизировать окно методом DoModal ():

```
afx_msg void WINDOW:: OnLButtonDown(UINT flags, CPoint loc)
{
   MY_DIALOG TheDialog( (LPTSTR) IDD_DIALOG1 , this);
   TheDialog.DoModal ();
};
```

6. Откомпилировать и выполнить приложение. Тестировать работу приложения.

Задание 3. Повторить задание 2, использовав автоматический каркас MFC, создаваемый мастером AppWizard (exe).

Задание 4. MFC-приложение на базе ТКП с диалоговым окном в качестве главного. *Создать MFC-приложение на базе ТКП с оконным интерфейсом из диалогового окна в качестве главного. Диалоговое окно содержит системное меню и кнопку “Справка”, при нажатии на которую выводится справочная информация о приложении. Завершение работы приложения производится соответствующей кнопкой системного меню диалогового окна.*

Порядок разработки (выполнения) задания.

1. Изучить результаты проектирования приложения (лекция 10, параграф “Диалоговое окно в качестве главного окна”).

2. Подготовить каркас MFC-приложения с файлом ресурсов. Создать типовой каркас приложения. Создать файл описания ресурсов и подключить его к приложению.

3. Создать новый ресурс - диалоговое окно. Добавить ресурс к проекту. Отредактировать внешний вид окна, настроить свойства окна.

4. Описать пользовательский класс MY_DIALOG для создания экземпляра главного окна

```
class MY_DIALOG : public CDialog
{
public:
   MY_DIALOG(char *DialogName, CWnd *Owner): CDialog(DialogName, Owner) { };
     afx_msg void OnOK ( );
     DECLARE_MESSAGE_MAP()
};
```

и карту сообщений

```
BEGIN_MESSAGE_MAP(MY_DIALOG, CDialog)
   ON_COMMAND(IDOK, OnOK)
END_MESSAGE_MAP()
```

Переопределить обработчик

```
afx_msg void MY_DIALOG::OnOK()
{
     MessageBox("Информация о приложении","О программе");
};
```

5. Подключить диалоговое окно к приложению – оно будет запускаться автоматически при запуске приложения. Для этого можно использовать конструктор приложения BOOL InitInstance(). Необходимо создать в нем экземпляр класса MY_DIALOG с именем TheDialog, инициализировать его параметрами (обликом) ресурса типа “диалоговое окно” с идентификатором ID = IDD_DIALOG1. Визуализировать и активизировать окно методом DoModal() как показано ниже

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

BOOL APPLICATION::InitInstance()
{    MY_DIALOG TheDialog((LPTSTR) IDD_DIALOG1 , NULL);
     TheDialog.DoModal();
     return TRUE;  } .
```

6. Откомпилировать и выполнить приложение.

Задание 5. Повторить задание 4, использовав автоматический каркас MFC, создаваемый мастером AppWizard (exe). Запустите приложение. Изучите его состав и функциональные возможности. Для этого запустите мастер MFC (например, AppWizard (exe)).

Выберите приложение, основанное на ДО. Остальные опции сохраните со значениями по умолчанию.

Задание 6. MFC-приложение с диалоговым окном в качестве главного.

Диалоговое окно содержит окошки редактирования для ввода-вывода данных. *Создать приложение для многократного ввода числовых значений и их вывода с противоположным знаком.*

Порядок разработки (выполнения) задания.

1. Изучить результаты проектирования приложения (лекция 10, параграф “Диалоговое окно в качестве главного - с окошком редактирования”).

2. Создать типовой каркас приложения, файл описания ресурсов и подключить его.

3. Создать новый ресурс - диалоговое окно. Добавить ресурс к проекту. Отредактировать внешний вид окна, настроить свойства окна.

Для поля эхо-вывода также используется окошко (поле) редактирования с ID – IDC_EDIT2.

Настройки окошка (вкладки General и Styles) должна быть выполнена с учетом использования этого элемента управления только в режиме вывода - отображения (read-only) данных.

Настройка свойств группирующего элемента (рамки Group Box) и статичного поля для отображения стрелки показаны ниже (рисунки 91, 92).

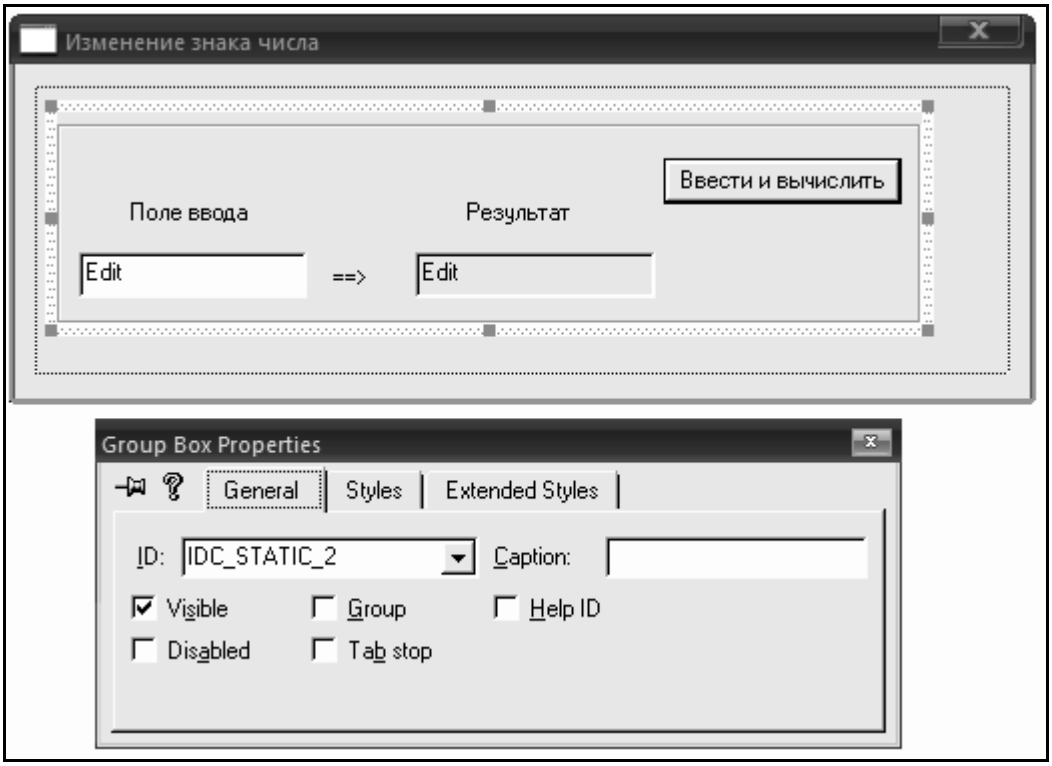


Рисунок 91 - Окно свойств группирующего ЭУ

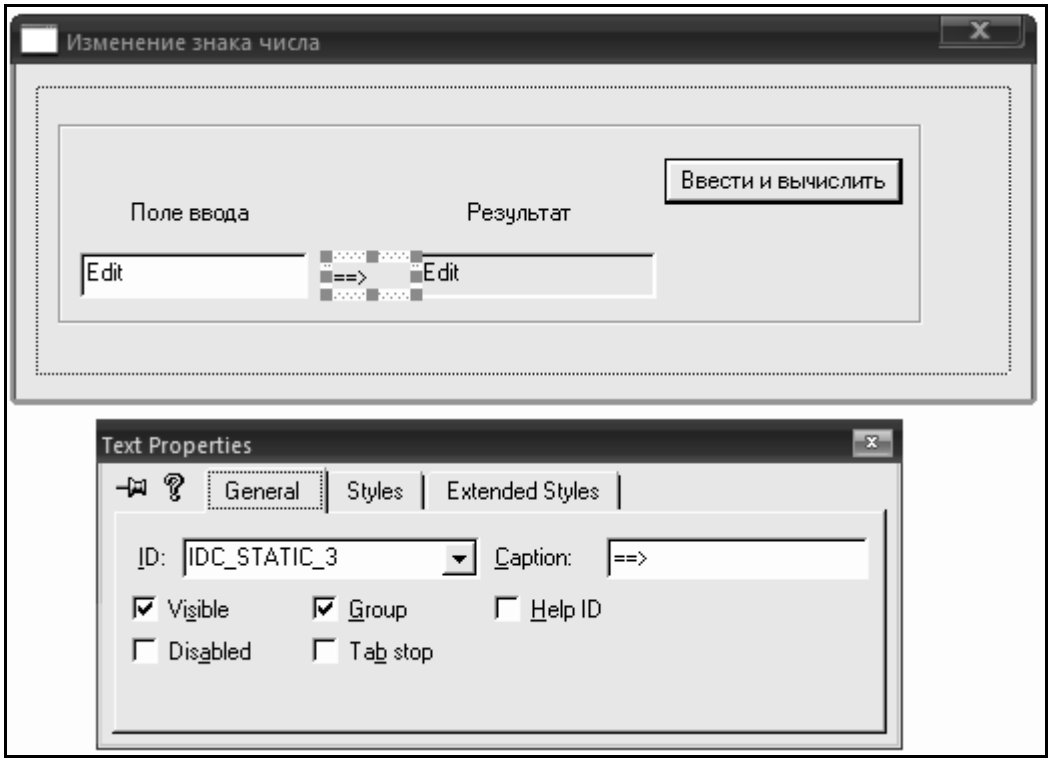


Рисунок 92 - Окно свойств статичного поля

Фрагмент содержимого ресурсного файла main.rc приведен ниже

```
//////////////////////////////////////////////////////////////////////////////
// Dialog
IDD_DIALOG1     DIALOG              DISCARDABLE  0, 0, 338, 103
STYLE               DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION          "Изменение знака числа"
FONT                8, "MS Sans Serif"
BEGIN
    DEFPUSHBUTTON   "Ввести и вычислить", IDOK,216,29,79,14
    EDITTEXT            IDC_EDIT1,21,58,76,14,ES_AUTOHSCROLL
    EDITTEXT            IDC_EDIT2,133,58,80,14,ES_AUTOHSCROLL | ES_READONLY
    CTEXT               "Поле ввода",IDC_STATIC_1,21,41,73,8
    CTEXT               "Результат",IDC_STATIC_2,129,41,76,8
    LTEXT               "==>",IDC_STATIC_3,105,61,26,8
    GROUPBOX            "",IDC_STATIC_2,14,15,288,65
END
//////////////////////////////////////////////////////////////////////////////
    ……..
```

4. Описать пользовательский класс MY_DIALOG для создания экземпляра главного окна как производный от библиотечного класса MFC CDialog и обслуживающий диалоговое окно IDD_DIALOG1.

Все пользовательские переменные, используемые для организации ввода-вывода значений, можно описать как глобальные переменные, но т.к. они используются только активным окном (class MY_DIALOG), то инкапсулируем их именно в класс окна в его открытую секцию. Например, используем переменные char InputStr[80], float FloatNumber, char OutputStr[80] соответственно для хранения введенного значения в строковом и числовом форматах и выводимого результата в строковом формате.

В отличие от предыдущих примеров добавить в описание метод OnInitDialog( ), а также перегрузить обработчик afx_msg void OnOK( )

```
class MY_DIALOG : public CDialog
{
public:
      char  InputStr[80];
      char  OutputStr[80];
      float  FloatNumber;
      MY_DIALOG(char *DialogName, CWnd *Owner): CDialog(DialogName, Owner) { };
      BOOL OnInitDialog ( );
      afx_msg void OnOK ( );
      DECLARE_MESSAGE_MAP()
};          .
```

Описать очередь сообщений

```
BEGIN_MESSAGE_MAP (MY_DIALOG,  CDialog)
    ON_COMMAND ( IDOK, OnOK)
END_MESSAGE_MAP( )          .
```

Описать метод OnInitDialog(), предназначенный для инициализации окошек редактирования – для задания их первоначального содержимого в момент вывода диалогового окна на экран

```
BOOL MY_DIALOG::OnInitDialog ( )
{
      CDialog::OnInitDialog ( );
      CEdit *pEditBox1 = (CEdit *) CDialog::GetDlgItem ( IDC_EDIT1 );
      pEditBox1 -> SetWindowText ( "Введите число" );
      return TRUE;
};                .
```

Описать обработчик сообщений afx_msg void OnOK( ), осуществляющий основные пользовательские действия по вводу значения в строковом формате как char InputStr[80], хранению значения в числовом формате как float FloatNumber и выводе результата в строковом формате как char OutputStr[80] соответственно

```
afx_msg void MY_DIALOG::OnOK ( )
{
      CEdit *pEditBox1 = ( CEdit * ) CDialog::GetDlgItem ( IDC_EDIT1 );
       //== ввод значения в виде строки
       pEditBox1 -> GetWindowText ( InputStr, sizeof InputStr-1 );
       pEditBox1 -> SetWindowText ( "      " );
       //== преобразование значения в тип float
       FloatNumber = atof ( InputStr );
       // == < ФРАГМЕНТ ПО ИСПОЛЬЗОВАНИЮ FloatNumber >
       FloatNumber = - FloatNumber;
       //== преобразование значения из типа float в строку
       _gcvt (FloatNumber, 15, OutputStr);
```

```
    CEdit *pEditBox2 = ( CEdit * ) CDialog::GetDlgItem( IDC_EDIT2 );
    //== вывод значения в виде строки
    pEditBox2 -> SetWindowText ( OutputStr );
    //== передача окошку ввода управления (фокуса)
    pEditBox1 -> SetFocus ( );
};  .
```

5. Подключить диалоговое окно к приложению – оно должно запускаться автоматически при инициализации пользовательского приложения в качестве интерфейсного окна вместо главного окна. Для этого необходимо скорректировать содержимое функции InitInstance (). А именно - необходимо создать диалоговое окно, например, как экземпляр TheDialog класса MY_DIALOG, инициализировав его параметрами облика (стиля) ресурса с идентификатором ID = IDD_DIALOG1. А затем необходимо провести визуализацию и активизацию окна как объекта TheDialog:

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

BOOL APPLICATION::InitInstance()
{    MY_DIALOG  TheDialog( (LPTSTR) IDD_DIALOG1, NULL);
     TheDialog.DoModal( );
     return TRUE;   }  .
```

6. Откомпилировать и выполнить приложение.

ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОГО ВЫПОЛНЕНИЯ. 1. Создать аналогичное приложение с запуском диалога по сообщению WM_PAINT, по нажатию клавиши клавиатуры и по нажатию клавиши мыши. 2. Создать приложение с диалоговым окном в качестве главного для вычисления квадратов вводимых значений. Результат выводить в отдельном диалоговом окне по нажатию соответствующей кнопки (например, Результат). После просмотра результата передавать управление исходному окну. 3. Создать приложение с диалоговым окном в качестве главного для вычисления кубов вводимых значений. Результат выводить в отдельном окне типа главного по нажатию соответствующей кнопки (например, Результат). После просмотра результата передавать управление исходному окну. 4. Создать MFC-приложение на базе ТКП для ввода массива вещественных числовых значений и их суммирования (рисунок 93). В поле ввода диалогового окна задается число, а ввод его в приложение осуществляется по нажатии клавиши ОК или Enter. Суммирование и вывод результата в поле ввода по кнопке СЛОЖИТЬ МАССИВ, а сброс системы для обработки следующего массива по кнопке СЛОЖИТЬ МАССИВ. Выход из приложения осуществляется по нажатии клавиши Esc или кнопки Cancel.

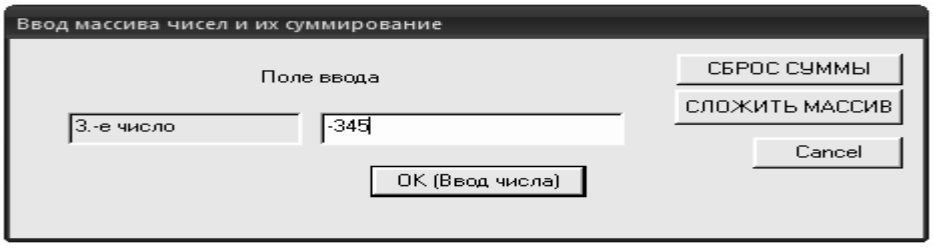


Рисунок 93 - Вид интерфейса

5. Создать MFC-приложение на базе ТКП с оконным интерфейсом из диалогового окна в качестве главного (рисунки 94, 95). Диалоговое окно содержит кнопку ОК и кнопку Cancel. При нажатии кнопки ОК пользователю предлагается загрузить следующее диалоговое окно такого же типа (класса) поверх текущего. Это позволяет создавать много диалоговых окон. При этом управление передается каждому следующему окну.

При нажатии кнопки Cancel текущее окно исчезает, а управление автоматически передается его родителю – предыдущему окну.

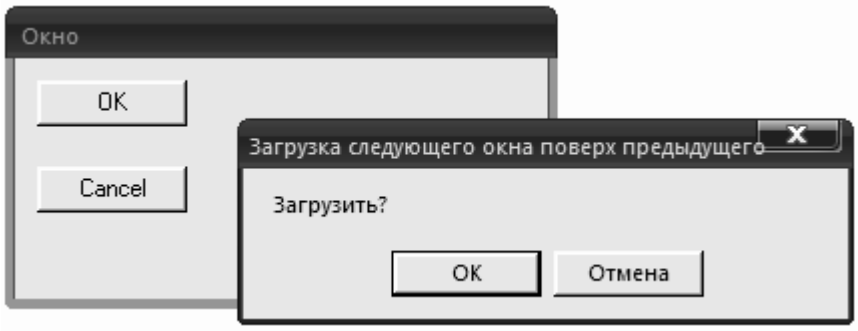


Рисунок 94 - Вид интерфейса

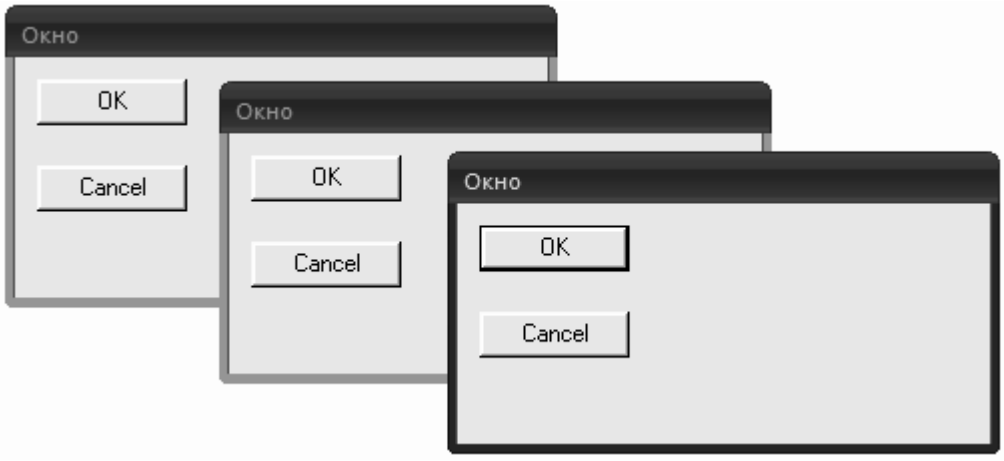


Рисунок 95 - Вид интерфейса

Задание 7. Выполнить самостоятельное задание п. 4.

Задание 8. Выполнить самостоятельное задание п. 5.

Задание 9. Изучить ресурс - меню (лекция 11, параграф “Использование пользовательского меню”).

Задание 10. MFC-приложение, содержащее пользовательское меню. *Создать приложение на базе ТКП с пользовательским меню. При выборе каждого конечного пункта менюдолжно выводиться подтверждающее сообщение. Завершение приложения производится закрытием главного окна либо выбором первого пункта меню. Предполагается, что визуализация окна с меню должна происходить автоматически каждый раз при запуске приложения.*

Порядок разработки (выполнения) задания.

1. Изучить результаты проектирования приложения (лекция 11, параграф “Использование пользовательского меню - приложение с пользовательским меню”).

2. Подготовить каркас MFC-приложения с файлом ресурсов.

2.1. Создать типовой каркас приложения.

2.2. Создать и подключить файл описания ресурсов к приложению посредством команды #include "resource.h". Соответственно после создания файла ресурсов *.rc в папке проекта появятся файлы main.rc, resource.h. В главном меню в пункте View активизируются пункты меню, связанные с работой с классами и ресурсами: ClassWizard, ID=ResourceSymbols, ResourceIncludes. Следует выполнить приложение и убедиться в его работоспособности.

3. Создать новый ресурс - меню.

3.1. Для этого надо добавить ресурс к проекту командой главного меню Insert-Resource-Menu и создать меню, пункты меню в редакторе ресурсов. Состав ресурсов представлен на рисунке 96.

Рисунок 96 - Ресурсы приложения

3.2. Настроить свойства меню и его пунктов, используя окна свойств. Окно свойств первого пункта меню представлено (рисунок 97) - надо задать название пункта и установить ранее определенные свойства. В качестве дескриптора этого пункта задано - IDM_MenuItem_1.

Рисунок 97 - Окно свойств пункта меню (Пункт_1)

Далее надо установить значения в окне свойств второго пункта меню (рисунок 98). Пункт не является конечным, поэтому дескриптор отсутствует.

Рисунок 98 - Окно свойств пункта меню (Пункт_2)

Окно свойств пункта меню 2.1 представлено на рисунке 99. Здесь для отметки пункта флажком устанавливается свойство Checked. Свойства других пунктов меню настраиваются аналогично

Рисунок 99 - Окно свойств пункта меню (Пункт_2.1)

При выполнении приложения файлы автоматически обновятся. Заголовочный файл описаний (resource.h) можно увидеть, использовав пункты главного меню View ID=ResourceSymbols. Если произвести компиляцию приложения в настоящем виде, то при выполнении приложения меню не будет визуализировано, так как оно как ресурс пока существует отдельно от приложения. Содержимое ресурсного файла (здесь main.rc), находящегося в папке проекта, можно просмотреть, например, в текстовом редакторе Word. Фрагмент содержимого ресурсного файла для рассматриваемого приложения приведен ниже.

```
//Microsoft Developer Studio generated resource script.
#include "resource.h"
...
// Menu
IDR_MENU1               MENU DISCARDABLE
BEGIN
    MENUITEM        "Пункт_1_(EXIT)",   IDM_MenuItem_1
    POPUP           "&Пункт_2"
    BEGIN
        MENUITEM    "П&ункт_2.1",       IDM_MenuItem_2_1, CHECKED
        MENUITEM    SEPARATOR
        POPUP       "Пун&кт_2.2"
        BEGIN
            MENUITEM "Пункт_2.2.1",     IDM_MenuItem_2_2_1
        END
    END
END   .
```

4. Далее следует подключить меню к приложению – к соответствующему окну приложения, используя 6-й параметр метода create. Пусть имя ресурса “меню” (его дескриптор) определено в файле resource.h командой #define как #define IDR_MENU1 101, то есть можно использовать или номер 101 или его макроопределение IDR_MENU1. Для подключения меню в конструкторе класса ОКНО следует указать

ОКНО::*ОКНО* ( )

```
{
…
    Create ( NULL, "Приложение с пользовательским меню",
             WS_OVERLAPPEDWINDOW,  rectDefault, NULL,  (LPTSTR) 101 );
```

или

```
    Create ( NULL, "Приложение с пользовательским меню",
             WS_OVERLAPPEDWINDOW,  rectDefault, NULL,  (LPTSTR) IDR_MENU1 );
 …  }   .
```

В результате меню будет загружаться автоматически при запуске приложения, однако пункты меню будут не активными (изображаются “бледно”).

5. Далее необходимо настроить класс ОКНО (здесь WINDOW) на работу с пользовательским меню.

5.1. В описание класса окна для каждого конечного пункта меню надо вставить прототипы указанных ранее функций-обработчиков сообщений события “выбор пункта меню”.

5.2. Описать функции-обработчики. Например, обработчик первого пункта меню

```
afx_msg void WINDOW:: OnMenuItem_1 ( )
{    MessageBox ( "пункт меню 1 ", " Выбран  " );
     SendMessage ( WM_CLOSE );  };    .
```

5.3. Включить чувствительность класса к сообщениям события “выбор пункта меню” путем внесения в карту сообщений окна следующих макрокоманд

```
BEGIN_MESSAGE_MAP(WINDOW,CFrameWnd)
     ON_COMMAND( IDM_MenuItem_1, OnMenuItem_1 )
     ON_COMMAND( IDM_MenuItem_2_1, OnMenuItem_2_1 )
     ON_COMMAND( IDM_MenuItem_2_2_1, OnMenuItem_2_2_1 )
END_MESSAGE_MAP()
```

6. Откомпилировать и выполнить приложение.

Задание 11. MFC-приложение, управляемое пользовательским меню, использующее окна и позволяющее вводить и выводить строковые данные.

Создать приложение с пользовательским меню, предназначенное для многократного ввода строк. При выборе пункта меню Enter должно выводиться диалоговое окно для ввода новой строки. При выборе пункта Display должна отображаться последняя введенная строка. Завершение работы приложения производится закрытием главного окна либо выбором пункта меню Exit.

Порядок разработки (выполнения) задания.

1. Изучить результаты проектирования приложения (лекция 11, параграф “Использование пользовательского меню - приложение с пользовательским меню и ДО”).

2. Необходимо создать каркас MFC-приложения с файлом ресурсов, обеспечить подключение заголовочных файлов и описание глобальной переменной для работы с данными

```
#include <afxwin.h>
#include <iostream>
#include "resource.h"
```

```
using namespace std;
char String[80];   .
```

3. Создать ресурсы приложения.

3.1. Создать ресурс – меню, настроить свойства меню и его пунктов.

3.2. Создать новый ресурс – диалоговое окно, настроить свойства окна и его элементов. Фрагмент содержимого ресурсного файла resource.h описан ниже:

```
//Microsoft Developer Studio generated resource script.
#include "resource.h"
…
// Dialog
IDD_DIALOG DIALOG DISCARDABLE  0, 0, 171, 55
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Окно ввода"
FONT 8, "MS Sans Serif"
BEGIN
   DEFPUSHBUTTON   "Enter",ID_ENTER,97,18,50,16
   EDITTEXT       IDC_EDIT,19,16,58,17,ES_AUTOHSCROLL
END
…
// Menu
IDR_MENU MENU DISCARDABLE
BEGIN
   MENUITEM "Exit",                    IDM_EXIT
   MENUITEM "Enter",                    IDM_ENTER
   MENUITEM "Display",                   IDM_DISPLAY
END
…
```

4. Необходимо подключить меню (дескриптор IDR_MENU) к приложению – к главному окну приложения (класс ГЛАВНОЕ_ОКНО). Для этого в конструкторе класса ГЛАВНОЕ_ОКНО следует указать

```
ГЛАВНОЕ_ОКНО:: ГЛАВНОЕ_ОКНО ( )
{  …
      Create ( NULL, "Обработка числа", WS_OVERLAPPEDWINDOW, rectDefault,
            NULL, (LPTSTR) IDR_MENU );
}   .
```

5. Далее надо настроить классы приложения ГЛАВНОЕ_ОКНО и ДИАЛОГОВОЕ_ОКНО.

5.1. Для настройки класса ДИАЛОГОВОЕ_ОКНО (DIALOG_WINDOW) в него следует включить прототипы функций-обработчиков сообщений события “выбор пункта меню”

```
class ДИАЛОГОВОЕ_ОКНО: public CDialog
{
public:
    ДИАЛОГОВОЕ_ОКНО (char *DialogName, CWnd *Owner):
          CDialog (DialogName, Owner) {   };
```

```
    afx_msg void OnEnter ( );
    DECLARE_MESSAGE_MAP()
};  ,
```

описать функцию-обработчик, например, как

```
afx_msg void ДИАЛОГОВОЕ_ОКНО::OnEnter( )
{
    CEdit *ptr = (CEdit *) GetDlgItem(IDC_EDIT);
    ptr->GetWindowText(String, sizeof String-1);
};
```

и включить чувствительность класса к соответствующим сообщениям

```
BEGIN_MESSAGE_MAP(ДИАЛОГОВОЕ_ОКНО, CDialog)
    ON_COMMAND(ID_ENTER, OnEnter)
END_MESSAGE_MAP()   .
```

5.2. Для настройки класса ГЛАВНОЕ_ОКНО (WINDOW) на работу с пользовательским меню в него следует включить прототипы функций-обработчиков сообщений события “выбор пункта меню”

```
class ГЛАВНОЕ_ОКНО: public CFrameWnd
{
public:
    ГЛАВНОЕ_ОКНО ( );
    afx_msg void OnExit( );
    afx_msg void OnEnter( );
    afx_msg void OnDisplay( );
    DECLARE_MESSAGE_MAP()
};   ,
```

описать функции-обработчики, например

```
afx_msg void ГЛАВНОЕ_ОКНО::OnEnter( )
{
    ДИАЛОГОВОЕ_ОКНО  MyDataEnterDialog((LPTSTR)IDD_DIALOG ,this);
    MyDataEnterDialog.DoModal( );
};

afx_msg void ГЛАВНОЕ_ОКНО::OnDisplay( )
{
    MessageBox(String,"Было введено: ");
};

afx_msg void ГЛАВНОЕ_ОКНО::OnExit( )
{
    int Answer;
    Answer = MessageBox("Quit the Program?","Exit", MB_YESNO
                        | MB_ICONQUESTION);
```

```
    if (Answer == IDYES)
            SendMessage(WM_CLOSE);
};
```

и включить чувствительность класса к соответствующим сообщениям

```
BEGIN_MESSAGE_MAP(ГЛАВНОЕ_ОКНО,CFrameWnd)
    ON_COMMAND(IDM_EXIT, OnExit)
    ON_COMMAND(IDM_ENTER, OnEnter)
    ON_COMMAND(IDM_DISPLAY, OnDisplay)
END_MESSAGE_MAP()   .
```

6. Создать класс ПРИЛОЖЕНИЕ (APPLICATION):

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

BOOL ПРИЛОЖЕНИЕ:: InitInstance()
{
    m_pMainWnd =  new WINDOW;
    m_pMainWnd -> ShowWindow(m_nCmdShow);
    m_pMainWnd -> UpdateWindow();
    return TRUE;
}  ,
```

объект приложения

```
ПРИЛОЖЕНИЕ МоеПриложение;   .
```

Далее необходимо приложение откомпилировать и выполнить.

Задание 12*. Модифицировать предыдущий пример: выводить квадрат значения в отдельном диалоговом окне.

Задание 13*. Модифицировать предыдущий пример: ввод-вывод значения выполнять в окне ВВОД ДАННЫХ. Режим работы этого окна задавать кнопками: по кнопке РЕЖИМ_ВВОД установить режим ввода, по кнопке РЕЖИМ_ВЫВОД установить режим вывода.

Задание 14. Изучить элемент управления список (лекция 11, параграф “Общие сведения о списках. Класс CListBox”).

Задание 15. MFC-приложение на базе ТКП для ведения списка записей – фамилий.

Исходный, а затем и модифицированный список фамилий должен отображаться в диалоговом окне в соответствующем списковом элементе управления (класса CListBox). Исходный список создается в момент инициализации диалогового окна последовательным добавлением к списку заранее определенных фамилий. Пользователь может осуществить выбор (выделение) фамилии одинарным либо двойным щелчком “мыши”. Выбранную (выделенную) фамилию здесь можно: - удалить; - использовать для создания и добавления к списку новой фамилии. Каждое из трех действий (выбор, удаление, добавление) должно сопровождаться выводом сообщения о его выполнении и требованием на подтверждение. Добавление фамилии здесь производится в упрощенном варианте и состоит во вводе в список копии уже содержащейся фа-

милии, выбранной одинарным или двойным щелчком, с добавлением к ней числового номера (например, Иванов1, Иванов5, …).

Порядок разработки (выполнения) задания.

1. Изучить результаты проектирования приложения (лекция 11, параграф “Общие сведения о списках. Класс CListBox - диалоговое окно со списком в качестве главного окна”).

2. Подготовить каркас MFC-приложения с файлом ресурсов. Для этого необходимо создать приложение на базе типового каркаса, создать файл описания ресурсов и подключить его к приложению.

3. Создать новый ресурс - диалоговое окно. Для этого необходимо в соответствии с видом графического интерфейса приложения (приведен выше) добавить ресурс - диалоговое окно, отредактировать его внешний вид, добавить необходимые элементы управления (список и кнопки), настроить свойства окна и его элементов. Примерное текстовое описание окна приведено ниже.

```
// Dialog
IDD_DIALOG1 DIALOG  DISCARDABLE  0, 0, 230, 103
STYLE  DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION  "Использование списка"
FONT  8, "MS Sans Serif"
BEGIN
   LISTBOX        IDC_LIST1, 15,15,81,37,LBS_SORT | LBS_NOINTEGRALHEIGHT |
                   WS_VSCROLL | WS_TABSTOP
   GROUPBOX       "Управление списком",IDC_STATIC,111,13,89,75
   PUSHBUTTON     "Добавить ФИО",IDC_BUTTON1,118,29,64,14
   PUSHBUTTON     "Удалить ФИО",IDC_BUTTON2,120,52,65,15
END     .
```

Здесь IDD_DIALOG1, IDC_LIST1, IDC_STATIC, IDC_BUTTON1, IDC_BUTTON2 - дескрипторы диалогового окна, списка, рамки и двух кнопок соответственно.

4. Необходимо описать пользовательский класс MY_DIALOG для создания экземпляра главного окна, например, как

```
class myDIALOG : public CDialog
{
public:
        myDIALOG(char *DialogName, CWnd *Owner): CDialog(DialogName, Owner)
         {       };
         BOOL OnInitDialog( );
         afx_msg void  OnToDisplaySelectedFIO( );
         afx_msg void  OnToAddFIO( );
          afx_msg void OnToDeleteFIO( );
         DECLARE_MESSAGE_MAP( )
};   .
```

Описать карту сообщений

```
BEGIN_MESSAGE_MAP(myDIALOG, CDialog)
```

```
        ON_LBN_DBLCLK(IDC_LIST1, OnToDisplaySelectedFIO)
        ON_COMMAND(IDC_BUTTON1, OnToAddFIO)
        ON_COMMAND(IDC_BUTTON2, OnToDeleteFIO)
END_MESSAGE_MAP( )   .
```

Описать метод OnInitDialog(), используемый здесь для инициализации списка – для задания его первоначального содержимого в момент вывода диалогового окна на экран.

```
BOOL myDIALOG::OnInitDialog( )
{
        CDialog::OnInitDialog();
        CListBox *PListBox = (CListBox *) GetDlgItem(IDC_LIST1);
        PListBox -> AddString("Иванов");
        PListBox -> AddString("Петров");
        PListBox -> AddString("Сидоров");
        PListBox -> AddString("Орлов");
        PListBox -> AddString("Мухов");
         return TRUE;
};      .
```

Описать обработчики сообщений

```
afx_msg void myDIALOG::OnToDisplaySelectedFIO( )
{
        char Str[80];
        MessageBox( "Выбор двойным щелчком","РАБОТАЕТ ФУНКЦИЯ" );
        CListBox *PListBox = (CListBox *) GetDlgItem( IDC_LIST1 );
        int i = PListBox ->GetCurSel( );
        PListBox ->GetText( i,Str );
        strcat( Str," - ваш выбор" );
        MessageBox( Str,"РЕЗУЛЬТАТ ВЫБОРА" );
};

afx_msg void myDIALOG::OnToAddFIO( )
{
        static int j=0;
        char Str[80];
        char Str1[80];
         char Str2[80];
        j++;
        MessageBox( "Добавить фамилию","РАБОТАЕТ ФУНКЦИЯ" );
        CListBox *PListBox = (CListBox *) GetDlgItem(IDC_LIST1);
        int i = PListBox -> GetCurSel( );
        if (i == LB_ERR )
        {
```

```
            MessageBox("Сделайте выбор","НАПОМИНАНИЕ", MB_OK |
                                MB_ICONWARNING);
        }
        else
        {
            PListBox ->GetText(i,Str);
            strcpy( Str1,Str );
            strcat( Str1," - ваш выбор" );
            MessageBox( Str1,"РЕЗУЛЬТАТ ВЫБОРА На ДОБАВЛЕНИЕ" );
            wsprintf( Str2,"%d",j );
            strcat( Str,Str2 );
            PListBox -> AddString( Str );
        };
};

afx_msg void myDIALOG::OnToDeleteFIO( )
{
        char Str[80];
        MessageBox( "Удалить фамилию","РАБОТАЕТ ФУНКЦИЯ" );
        CListBox *PListBox = (CListBox *) GetDlgItem( IDC_LIST1);
        int i = PListBox -> GetCurSel( );
        if  (i == LB_ERR )
        {
                MessageBox( "Сделайте выбор","НАПОМИНАНИЕ", MB_OK |
                                MB_ICONWARNING );
        }
        else
        {
                i = PListBox -> GetCurSel( );
                PListBox ->GetText( i,Str );
                strcat( Str," - удалять ?" );
                   if (MessageBox(Str,"РЕЗУЛЬТАТ ВЫБОРА НА УДАЛЕНИЕ",
                                MB_OKCANCEL) == IDOK)
                {
                        PListBox -> DeleteString( i );
                };
        };
};   .
```

5. Подключить диалоговое окно к приложению – оно должно запускаться автоматически при инициализации пользовательского приложения в качестве главного окна

```
class myAPPLICATION:public CWinApp
{
public:
        BOOL InitInstance( );
```

```
};

BOOL myAPPLICATION::InitInstance( )
{
        myDIALOG  TheDialog( ( LPTSTR ) IDD_DIALOG1,NULL );
        TheDialog.DoModal( );
        return TRUE;
}  .
```

6. Откомпилировать и выполнить приложение.

ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОГО ВЫПОЛНЕНИЯ. 1. Модифицировать приложение, добавив отображение в диалоговом окне выбранной строки, номера выбранной строки, а также числа строк в списке. 2*. Модифицировать предыдущее приложение для удаления только фамилий, выбранных двойным щелчком. 3*. Модифицировать приложение в части добавления в список новой введенной фамилии. Для этого предусмотреть в том же диалоговом окне сегмент с полем ввода, полем отображения ввода и кнопками управления вводом. Разработать соответствующуюдиаграмму состояний UML. 4*. Модифицировать предыдущее приложение, добавив проверку новой фамилии на совпадение с уже имеющимися и на корректность написания фамилии. 5. Модифицировать предыдущее приложение, использовав для ввода новой фамилии отдельное диалоговое окно. 6. Модифицировать предыдущее приложение, обеспечив автоматическое сохранение списка в файле при закрытии приложения и загрузку списка из файла при запуске приложения. Использовать один и тот же файл. 7.* Модифицировать предыдущее приложение, обеспечив автоматическое сохранение списка при закрытии приложения в заданном файле и загрузку списка при запуске приложения из заданного файла. 8.* Модифицировать предыдущее приложение, обеспечив автоматическое сохранение списка в заданном файле и загрузку списка из заданного файла в любой момент работы приложения. 9. Модифицировать приложение, реализовав поддержку обработки списка с помощью соответствующего пользовательского класса. Добавить поиск строки в списке по заданному образцу. 10*. Модифицировать приложение, реализовав систему меню для поддержки всех операций.

ЛАБОРАТОРНАЯ РАБОТА № 6. “Средства автоматизации разработки приложений

ЦЕЛЬ РАБОТЫ: 1. Изучение авто-каркасов приложений, средств создания и редактирования классов. 2. Изучение средств разработки окон, диалоговых окон, меню, элементов управления.

СОДЕРЖАНИЕ ОТЧЕТА: Описание результатов использования средств автоматизации: - диаграммы классов каркасов приложений и разработанных приложений; - автоматные диаграммы приложения. Описание особенностей разработки и управления графическим интерфейсом, ресурсами. Примечание - пункты, помеченные *, рассматривать как задания на самостоятельное выполнение и выполнять по указанию преподавателя.

ЗАДАНИЯ ДЛЯ ВЫПОЛНЕНИЯ

Задание 1. Создать приложение на базе каркаса MFC, аналогичном базовому ТКП. Приложение содержит:

1. Главное окно (ГО) с клиентской областью (КО) и меню (ГМ) с пунктами управления: - запуском модального окна ДО1, которое используется для ввода очередной строки; - запуском и разрушением немодального окна ДО2, которое нужно для вывода списка значений; - выводом последней введенной строки.

2. Модальное окно ДО1 для ввода очередной строки. Его функциональность обеспечивается классом DO1. Окну соответствует временный объект DO1 do1, создаваемый при его запуске соответствующим пунктом меню в обработчике этого пункта void CChildView::OnDialogsModal() { DO1 do1; … } . Создание и визуализация самого окна производится пунктом главного меню Dialogs-Modal.

3. Немодальное окно ДО2 вывода списка значений, которыми окно инициализируется. Его функциональность обеспечивается классом DO2. Окну соответствует объект DO2 do2, являющийся членом класса каркаса class CChildView: public CWnd и создаваемый при инициализации приложения. Создание и визуализация самого окна производится пунктом главного меню ГМ-Dialogs-Notmodal, а разрушение - ГМ-Dialogs-NotmodalDel.

Вид интерфейсных форм (ГО, ДО1, ДО2) приведен ниже (рисунки 100-102).

Требуется обеспечить следующие функции: 1. Ввод значения строки через модальное диалоговое окно ДО1 с последующим сохранением в пользовательском классе. 2. Отображение в списке немодального окна ДО2 результатов его инициализации. 3. Вывод последней введенной строки в клиентскую область окна по выбору команды главного меню или при перерисовке окна или нажатием левой кнопки мыши. 4. Изменение пунктов главного меню при управлении немодальным окном (обновление пунктов), препятствующее повторному созданию того же окна или его разрушению, если окно еще не создано.

Для запоминания состояния окна ДО2 используется член класса CChildView – переменная public: int m_NotModalState. Состояние ноль означает, что окно не создано. Состояние один означает, что окно создано (видимо или нет). Соответственно реализуется динамическая попеременная блокировка пунктов меню ГМ-Dialogs-Notmodal, ГМ-Dialogs-NotmodalDel. В начальном состоянии (m_NotModalState = 0) – блокируется пункт главного меню ГМ-Dialogs-NotmodalDel, а после создания окна значение m_NotModalState = 1 и блокируется пункт главного меню ГМ-Dialogs-Notmodal.

Далее необходимо внести изменения в приложение: 1*. Модифицировать модальное окно для ввода массива строк в одном сеансе. В качестве текущей строки для вывода в КО рассматривать последнюю введенную. 2*. Добавить пользовательский класса для поддержки работы с массивом строк. 3*. Использовать ЭУ список немодального окна для отображения всех введенных строк. 4*. Использовать список немодального окна для выбора строки из списка введенных в качестве текущей с возможностью ее вывода в КО при выборе пункта меню Print или щелчком левой клавиши мыши. 5*. Использовать ЭУ список немодального окна для выбора строки из списка введенных для последующего удаления из объекта пользовательского класса.

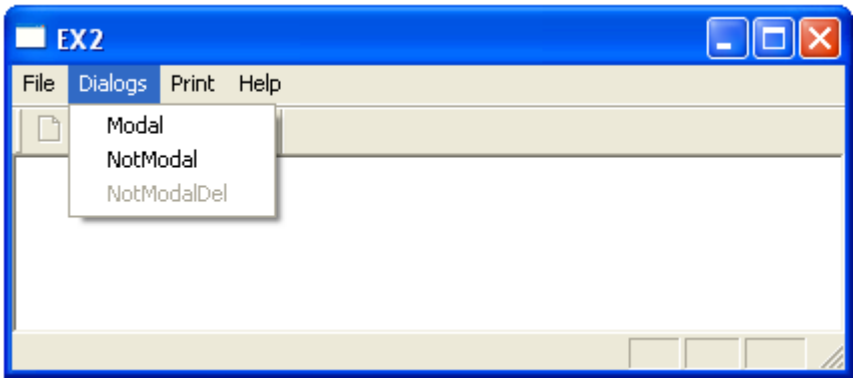


Рисунок 100 - Главное окно

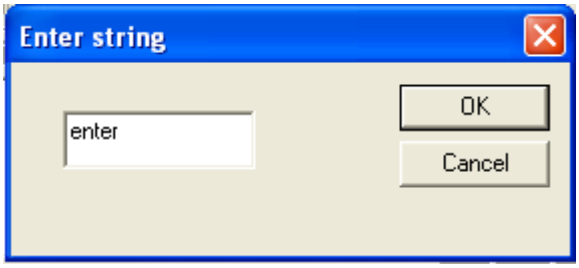


Рисунок 101 - Модальное ДО

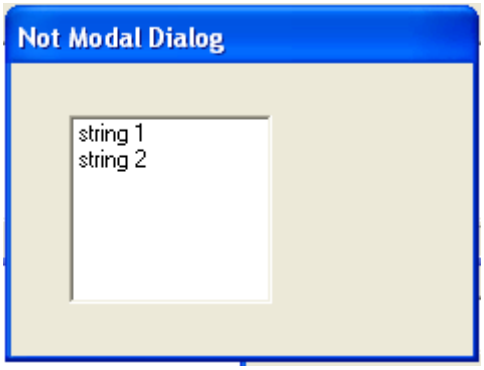


Рисунок 102 - Немодальное ДО

Состав классов приложения представлен ниже (рисунок 103).

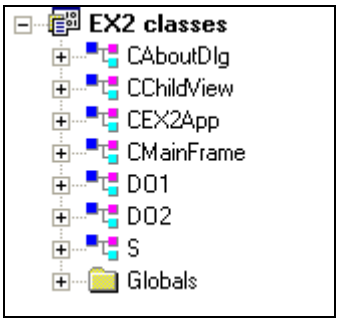


Рисунок 103 - Классы приложения

Структура классов приложения, включая классы диалоговых окон DO1, DO2 и пользовательского класса S, приведена ниже (рисунки 104-106).

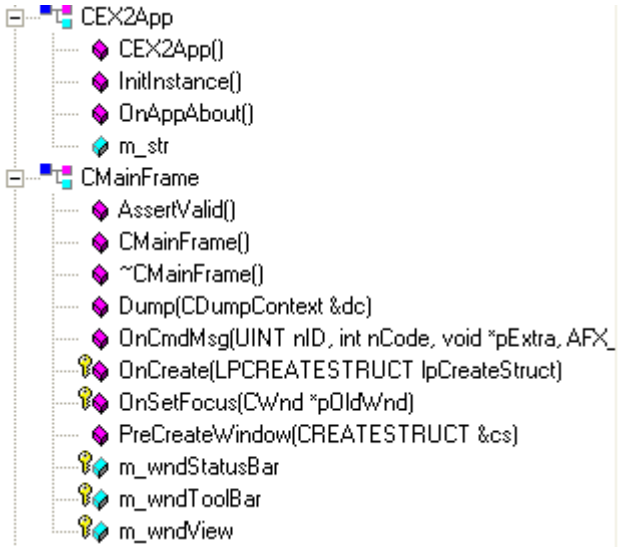


Рисунок 104 - Базовые классы ТКП

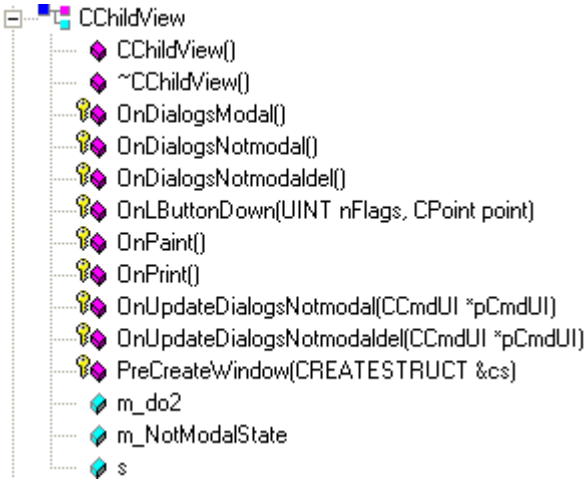


Рисунок 105 - Классы CChildView

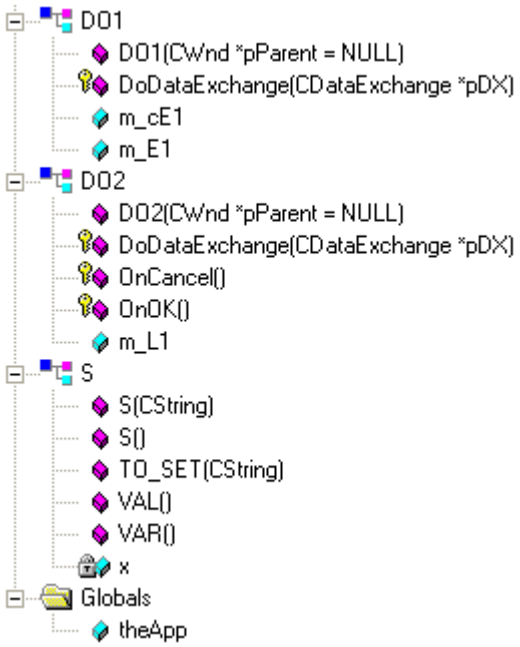


Рисунок 106 - Классы ДО, пользовательский класс S

Порядок разработки (выполнения) задания.

1. Создать каркас. Для этого запустите мастер MFC AppWizard (exe). В поле Project name введите название проекта приложения и щелкните кнопку ОК. В первом окне мастера выберите - однодокументное (Single Document Interface - SDI, флажок - Single document) и откажитесь от поддержки архитектуры Документ-Вид. Остальные опции сохраняем по умолчанию. Ниже (рисунок 107) приведен вид интерфейса каркаса.

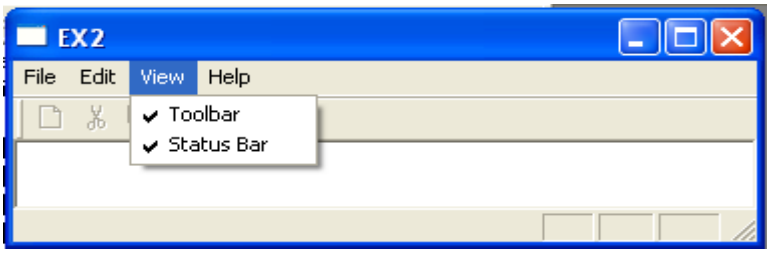


Рисунок 107 - Интерфейса каркаса

2. Редактировать меню. Выполняется с помощью редактора меню. Здесь меню нужно привести к заданному в задании виду.

3. Добавить обработчики командных сообщений меню. Вначале это могут быть пустые обработчики с подтверждением выбора соответствующего из пунктов. Здесь это обработчики пунктов с ID_PRINT, ID_DIALOGS_MODAL, ID_DIALOGS_NOTMODAL, ID_DIALOGS_NOTMODALDEL. Обработчики добавить мастером ClassWizard в class CChildView: public CWnd. Добавьте пустые обработчики для пунктов DialogsModal, DialogsNotmodal, DialogsNotmodaldel. А для пункта Print (рисунок 108), так как он нужен для вывода в КО новой строки из пользовательского класса, в обработчике пошлем требование на перерисовку КО. Саму строку надо выводить в методе OnPaint().

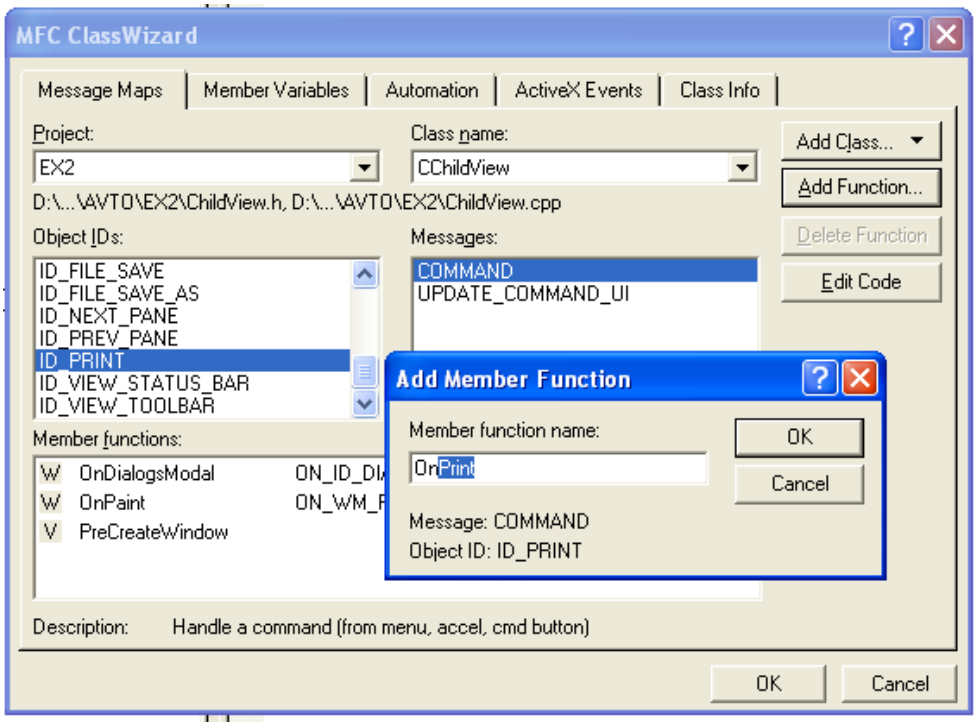


Рисунок 108 – Добавление обработчика

Соответственно код обработчика может выглядеть как void CChildView::OnPrint( ) { // TODO: Add your command handler code here InvalidateRect(NULL,TRUE); } .

4. Добавить обработчики обновления меню. Здесь следует управлять блокировкой пунктов меню для немодального окна - DialogsNotmodal, DialogsNotmodaldel. Добавьте в класс class CChildView:public CWnd переменную состояния немодального окна public: int m_NotModalState (рисунок 109). В конструкторе CChildView() установите начальное значение m_NotModalState= 0, в обработчике OnDialogsNotmodal() задайте m_NotModalState = 1, в обработчике OnDialogsNotmodaldel() задайте m_NotModalState = 0.

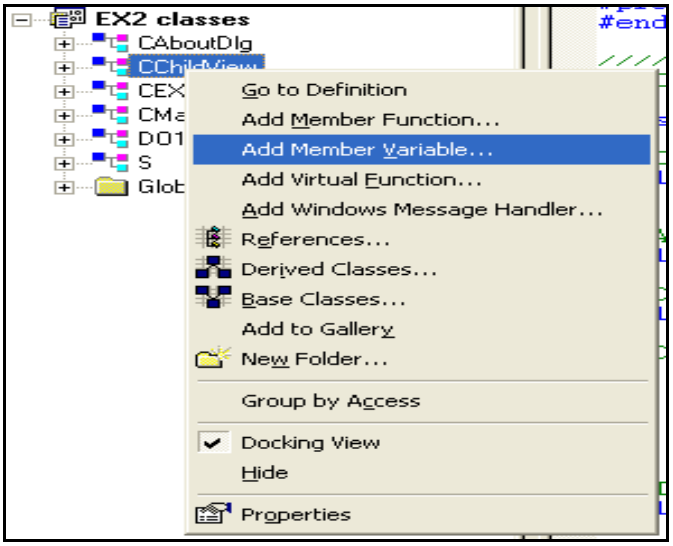


Рисунок 109 – Добавление атрибутов класса

Добавьте обработчики сообщений обновления меню (рисунок 110) (сообщение UPDATE_COMMAND_UI) для обоих пунктов меню (DialogsNotmodal и DialogsNotmodaldel).

Прототипы обработчиков: void CChildView::OnUpdateDialogsNotmodal(CCmdUI* pCmdUI) и void CChildView::OnUpdateDialogsNotmodaldel(CCmdUI* pCmdUI) .

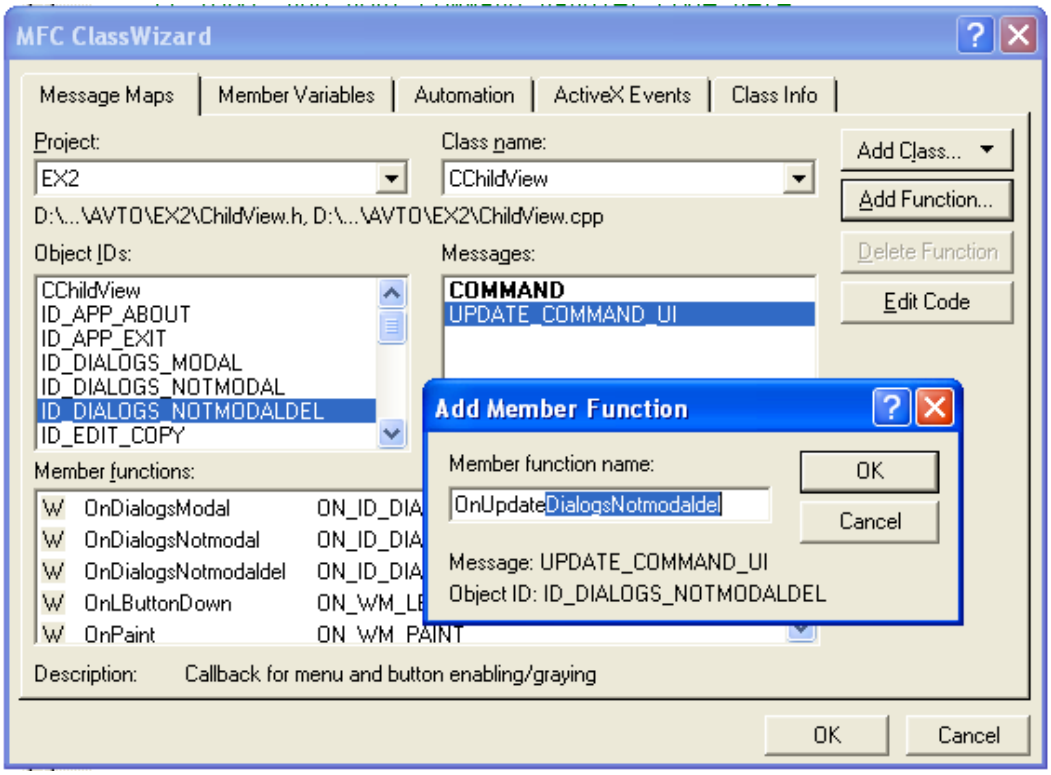


Рисунок 110 - Добавление обработчика

Добавьте необходимые коды в обработчики в соответствии с текстами ниже

```
void CChildView::OnUpdateDialogsNotmodal(CCmdUI* pCmdUI)
{    // TODO: Add your command update UI handler code here
     if (m_NotModalState == 1)  pCmdUI->Enable(FALSE);   }

void CChildView::OnUpdateDialogsNotmodaldel(CCmdUI* pCmdUI)
{    // TODO: Add your command update UI handler code here
     if (m_NotModalState == 0)  pCmdUI->Enable(FALSE);  }  .
```

Тогда меню высвечивается изначально с блокировкой пункта меню NotModalDel. Но после создания модального окна пунктом NotModal пункт меню NotModalDel разблокируется, а NotModal наоборот – станет не активным.

5. Добавить пользовательский класс. При этом место размещения кода класса может быть разным. Так его можно вставить в уже имеющиеся в проекте файлы *.cpp, *.h. Например, здесь его можно разместить в файлах описания класса CChildView (CChildView.cpp, CChildView.h). Описание пользовательского класса можно вставить и в специально созданные для этого в проекте новые файлы *.cpp, *.h. Для этого использу-

ется окно New, вкладка Files. Здесь в проекте надо: 1) создать пользовательский класс для хранения и обработки строки; 2) добавить к проекту два файла S.cpp, S.h (здесь S – имя класса для хранения и обработки строки); 3) разместить код в файлах.

Для этого: - открыть окно командой ГМ-project-AddToProjectFiles и создать оба указанных выше файла; - ввести код описания класса в S.h

```
class S
{
      CString x;
       public:  S(); S(CString); CString &VAR(); TO_SET(CString);
};
```

и код реализации в S.cpp

```
#include "stdafx.h"
#include "S.h"
S::S( ) {x="";}; S::S(CString TheX){x=TheX;}; CString &S::VAR(){return x;};
S::TO_SET(CString TheX){x=TheX;    };   .
```

6. Добавить в класс новые члены (методы, атрибуты). Здесь к классу S надо добавить метод CString VAL() - соответственно прототип CString VAL() и реализацию CString S::VAL() { return x; }. Для этого надо: - вызвать контекстное меню и выбрать команду добавления (рисунок 111); - описать метод и добавить код (рисунок 112). Аналогично можно добавлять атрибуты. Здесь надо добавить в класс CChildView объект пользовательского класса S s.

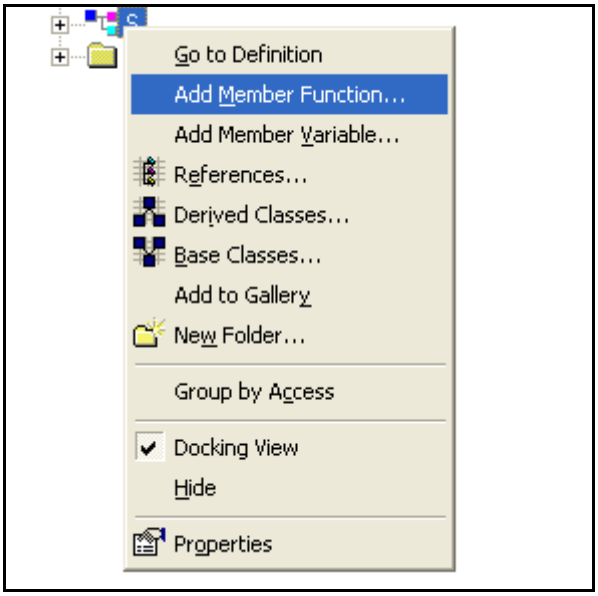


Рисунок 111 - Добавление методов класса

Рисунок 112 - Добавление методов класса

7. Добавить обработчики оконных сообщений Windows (они представлены перегруженными виртуальными функциями базового класса CWnd – с префиксом "On". В проекте надо добавить обработчик события - нажатие левой клавиши мыши. Указанное событие должно вызывать действия аналогичные выбору пункта ГМ-Print. Для этого необходимо: - открыть ClassWizard, нажав Ctrl-W (рисунок 113); - перейти на вкладку Message Maps; - выбрать в списке Class Name элемент CChildView (так обрабатывается сообщение, посланное классу, который инкапсулирует клиентскую область главного окна); - выбрать в списке Object IDs пункт CChildView; - в списке Messages выбрать сообщение WM_LBUTTONDOWN; - указать Add Function, а затем Edit Code; - так как обработчик будет использоваться для вывода в КО ранее введенной строки, сохраненной в пользовательском классе, то необходимо инициировать перерисовку КО. Соответственно надо добавить в обработчик команду InvalidateRect (NULL,TRUE)

```
afx_msg void CChildView::OnLButtonDown(UINT nFlags, CPoint point)
{ // TODO: Add your message handler code here and/or call default
    InvalidateRect(NULL,TRUE);  CWnd::OnLButtonDown(nFlags, point);  }   .
```

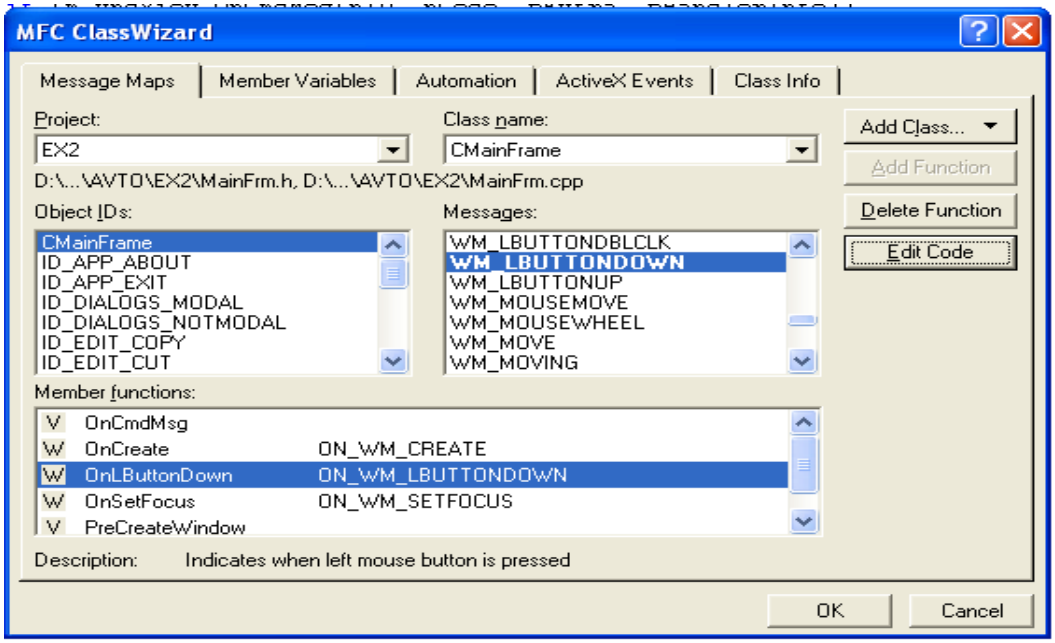


Рисунок 113 – Добавление оконного обработчика

8. Обеспечьте вывод в КО главного окна ранее введенной строки, сохраненной в пользовательском классе. В используемом каркасе используйте метод класса CChildView, подменяющий соответствующий метод OnPaint() базового класса CWnd.

```
void CChildView :: OnPaint( )
{    CPaintDC dc(this); // device context for painting
     // TODO: Add your message handler code here
     dc.TextOut( 1,1,s.VAR( ) );   }   .
```

9. Создать модальное окно Такое окно получает управление и удерживает его, пока пользователь не закроет окно. Здесь надо создать ДО с IDD_DIALOG1. Оно используется для ввода строки в окне редактирования с IDC_EDIT1 и помещения ее по кнопке с IDOK в переменной CString m_E1, связанной с ЭУ с IDC_EDIT1. Мможно использовать переменную m_cE1 (CEdit m_cE1) для управления окном редактирования. Функциональность ДО обеспечивается классом DO1, связанным с ресурсом IDD_DIALOG1. Окно создается при запуске обработчика пункта меню Modal. Тогда в обработчике в класса CChildView: - создается локальный объект do1 класса диалогового окна (DO1 do1); - выполняется внешняя инициализация ЭУ (do1.m_E1 = "enter"); - окно активизируется

```
DO1 do1;
do1.m_E1 = "enter";
int  nResponse = do1.DoModal( );   .
```

9.1. Создайте указанный выше ресурс – диалоговое окно с IDD_DIALOG1. 9.2. Создайте класс окна: создается с помощью мастера ClassWizard. Здесь нужно создать класс DO1, связанный с ресурсом – диалоговое окно с IDD_DIALOG1. Окна настройки мастера приведены ниже (рисунки 114, 115).

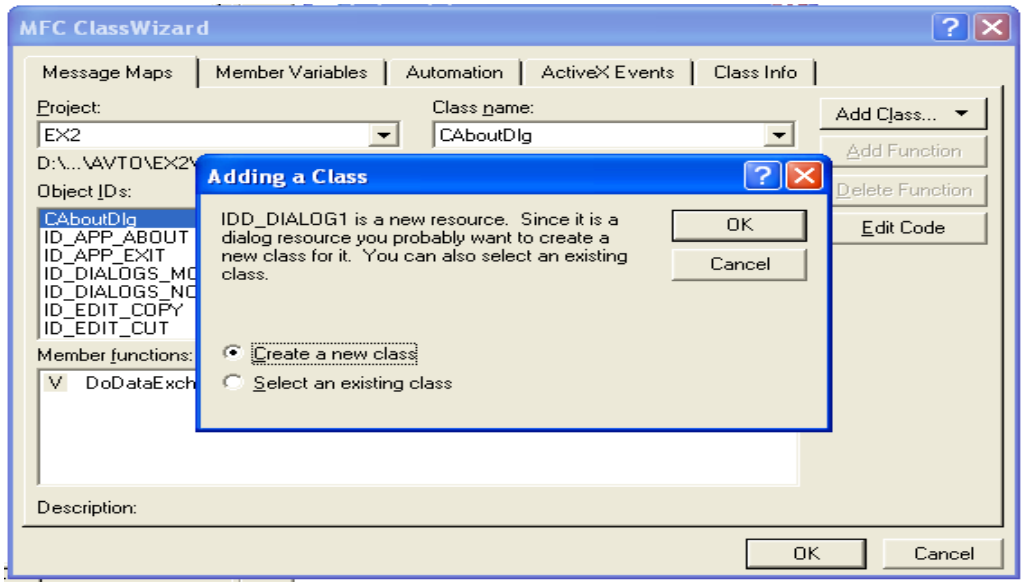


Рисунок 114 - Добавление класса

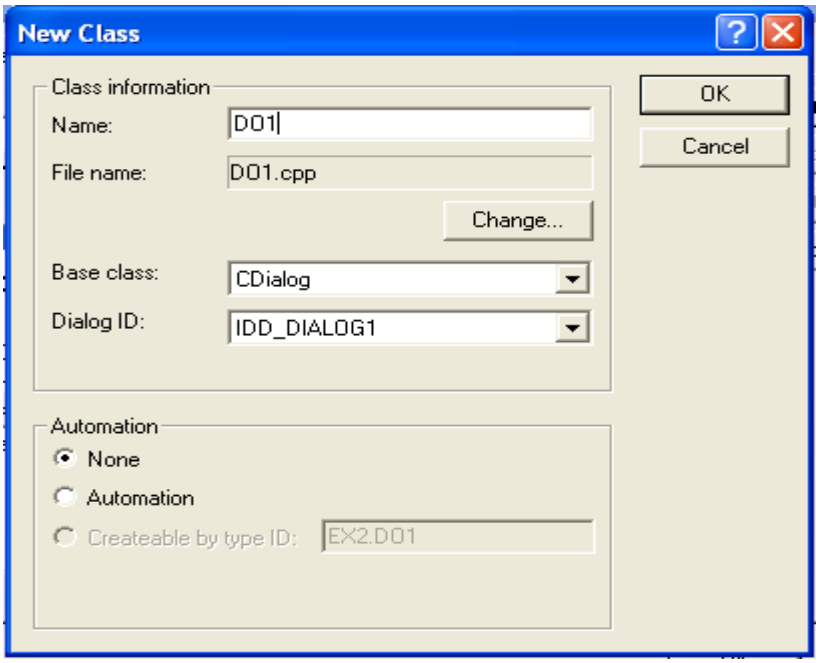


Рисунок 115 – Добавление класса

9.3. Добавить переменные для управления элементами окна (делается с помощью мастера ClassWizard). В проекте это переменные m_cE1 (CEdit m_cE1) для управления и m_E1 (CString m_E1) для сохранения значения окна редактирования, связанные с ЭУ с IDC_EDIT1 диалогового окна. Окна настройки мастера приведены ниже (рисунки 116, 117).

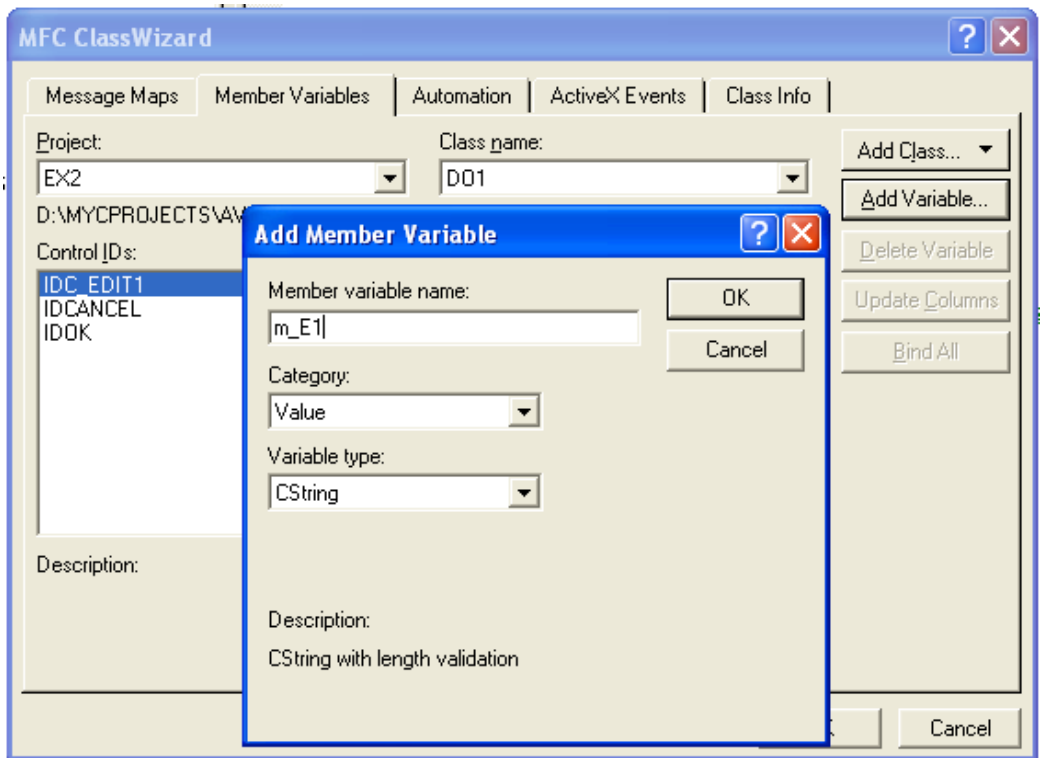


Рисунок 116 - Добавление атрибута класса

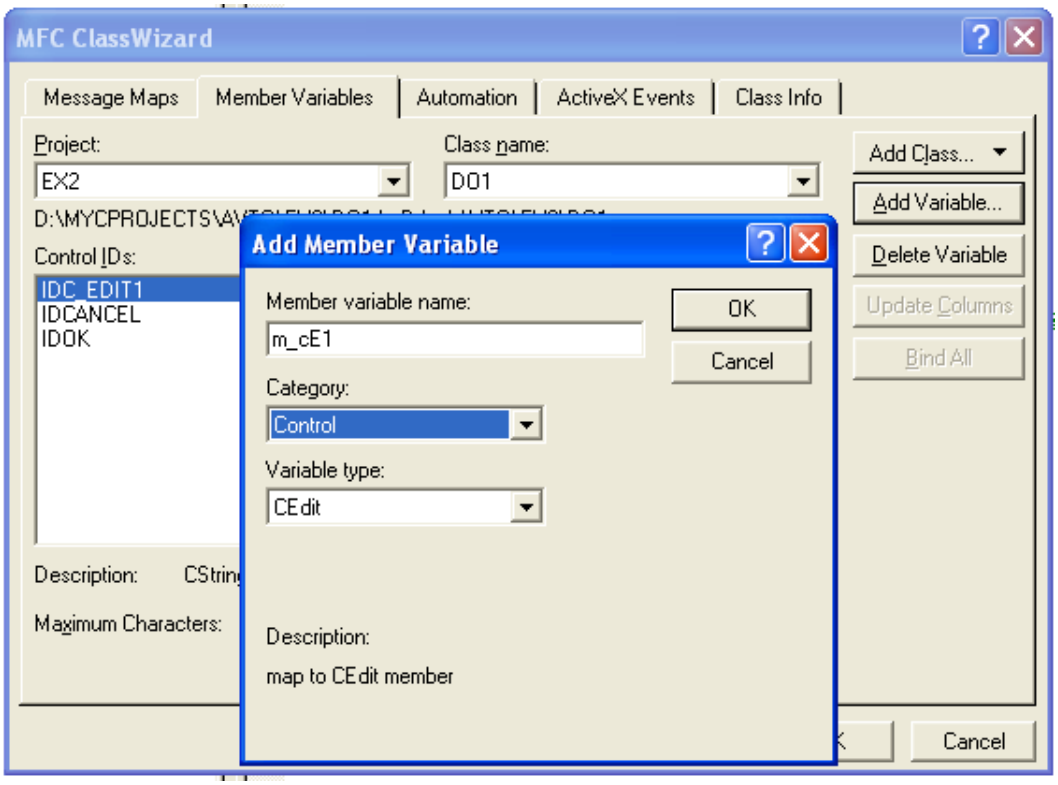


Рисунок 117 - Добавление атрибута класса

9.4. Организация запуска окна. Для запуска окна необходимо: - создать объект do класса ДО; - осуществить запуск и контролировать завершение и код завершения работы с ДО, например, как

```
int nResponse = do.DoModal( );
if (nResponse == IDOK)
{
        // TODO: Place code here to handle when the dialog is
        //  dismissed with OK
           …
}
else if (nResponse == IDCANCEL)
{
        // TODO: Place code here to handle when the dialog is
        //  dismissed with Cancel
           …
}  .
```

В проекте запуск ДО должен производиться в обработчике void CChildView::OnDialogsModal(). Для этого надо отредактировать код соответствующего обработчика как показано ниже

```
void CChildView::OnDialogsModal( )
```

```
        {
// TODO: Add your command handler code here
DO1  do1;  do1.m_E1 = "enter";  int nResponse = do1.DoModal( );
if ( nResponse == IDOK )
    {   MessageBox(do1.m_E1,"string");    }
else if ( nResponse == IDCANCEL )  {  MessageBox(do1.m_E1,"string");   }
         } .
```

9.5. Инициализация окна. Стандартный способ – использовать для этого метод OnInitDialog() класса диалогового окна. Другой способ инициализации – ”внешняя” инициализация, когда используются переменные класса ДО, связанные с ЭУ. Они устанавливаются нужными исходными значениями до момента запуска окна. При отсутствии метода OnInitDialog в пользовательском классе ДО его следует добавить мастером ClassWizard. После выбора команды Add Function окно примет вид (рисунок 118).

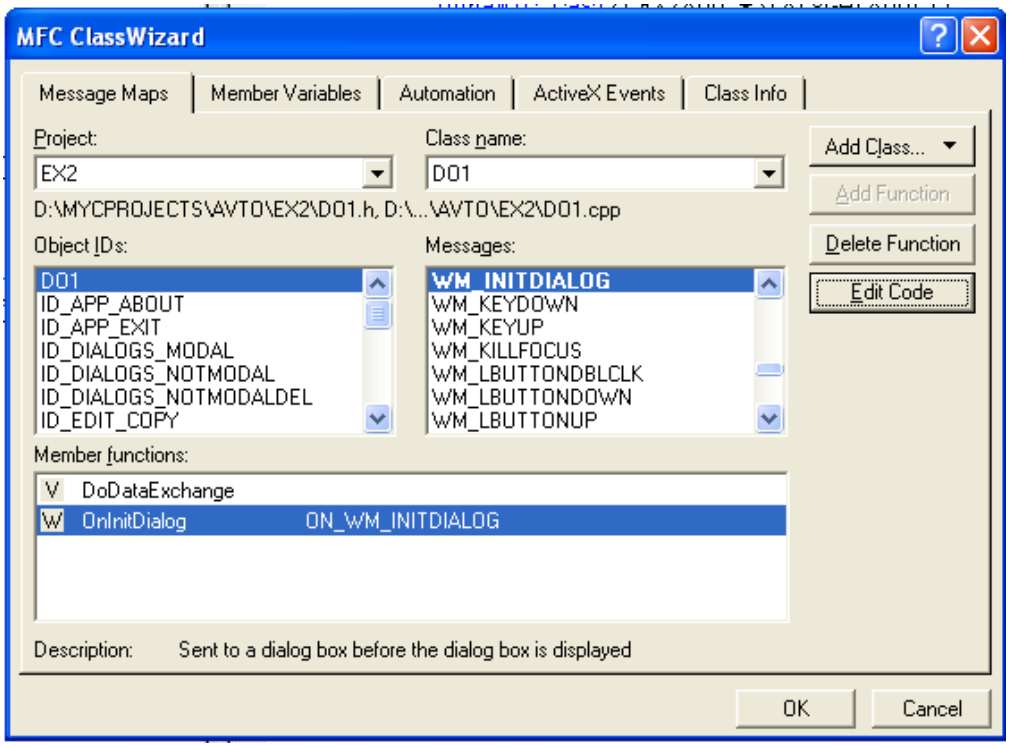


Рисунок 118 - Добавление обработчика для инициализации окна

После выбора команды Edit Code произойдет переход в описание кода обработчика, где и надо реализовать нужную инициализацию ДО (соответствующих ЭУ)

```
BOOL DO1::OnInitDialog( )
{
    CDialog::OnInitDialog( );
    // TODO: Add extra initialization here
    return TRUE;  }     .
```

10. Создать модальное окно. Немодальные окна не монополизирует управление приложением и, оставаясь открытыми, позволяют работать с несколькими окнами одновременно, перемещая фокус произвольным образом с одного окна на другое. Здесь надо создать немодальное ДО с IDD_DIALOG2. Оно не содержит кнопок с IDOK, IDCANCEL и системных меню. Используется для вывода списка значений в окне ЭУ список с IDC_LIST1. Следует использовать переменную m_L1 (CListBox m_L1), связанную с IDC_LIST1, для управления списком. Функциональность ДО обеспечивается классом DO2, связанным с ресурсом IDD_DIALOG2. Окну соответствует объект DO2 m_do2, являющийся атрибутом класса каркаса CChildView (class CChildView:public CWnd). Объект создается при создании и инициализации приложения.

Создание и визуализация самого окна производится пунктом главного меню Dialogs-Notmodal – запуском обработчика void CChildView::OnDialogsNotmodal(), а разрушение производится пунктом главного меню Dialogs-NotmodalDel – запуском обработчика void CChildView::OnDialogsNotmodaldel(). Соответственно в обработчике OnDialogsNotmodal выполняется создание и визуализация окна как m_do2.Create(IDD_DIALOG2); m_do2.ShowWindow(SW_SHOW); . А в обработчике OnDialogsNotmodaldel окно разрушается m_do2.DestroyWindow(); .

10.1. Создайте указанный выше ресурс – диалоговое окно с IDD_DIALOG2 и отмените системное меню.

10.2. Создайте класс окна (аналогично п. 9.2.) DO2, связанный с ресурсом – диалоговое окно с IDD_DIALOG2.

10.3. Модифицируйте класс окна с целью исключить возможность скрытия-удаления окна действиями, отличными от разрешенных. В проекте управление окном производится только через пункты главного меню. Соответственно необходимо: - добавить с помощью мастера ClassWizard обработчики нажатия кнопок с IDOK, IDCANCEL; - удалить в них стандартные действия, закомментировав вызов стандартных обработчиков, и удалить сами кнопки OK и Cancel

```
void DO2::OnCancel( ) { //CDialog::OnCancel( ); }
void DO2::OnOK( ) {//CDialog::OnOK( ); }   .
```

10.4. Добавить переменные для управления элементами окна (аналогично п. 9.3.). В проекте это переменная m_L1 (CListBox m_L1), связанная с IDC_LIST1, для управления списком. Используется для вывода списка значений в окне ЭУ список с IDC_LIST1.

10.5. Организация запуска и разрушения окна (выполняется аналогично п. 9.4.). Окно запускается, визуализируется-скрывается и разрушается как и окна типа главного методами Create, ShowWindow, DestroyWindow. Для этого надо создать объект m_do2 класса DO2 как атрибут класса CChildView, использовав, например, контекстное меню вкладки Классы как показано ниже (рисунок 119).

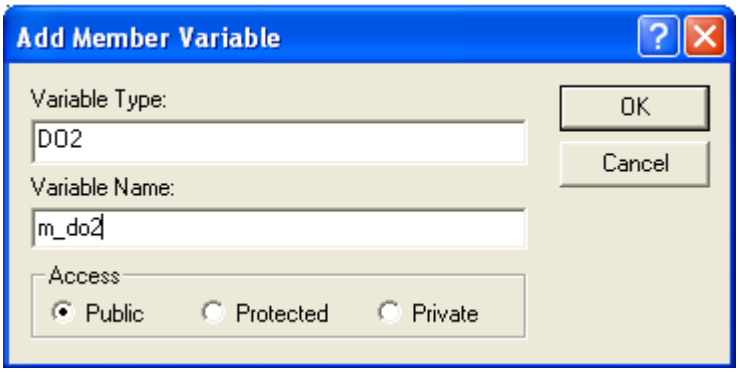


Рисунок 119 – Добавление объекта класса ДО

В проекте запуск окна должен производиться в обработчике как

```
void CChildView::OnDialogsNotmodal ( )
{     // TODO: Add your command handler code here
      m_do2.Create(IDD_DIALOG2);
      m_do2.ShowWindow( SW_SHOW );
      m_NotModalState = 1;        }.
```

Разрушение окна выполнять в обработчике как

```
void CChildView::OnDialogsNotmodaldel( )
{
      // TODO: Add your command handler code here
      m_do2.DestroyWindow( );
      m_NotModalState = 0;   }    .
```

10.7. Инициализация окна (выполняется аналогично п. 9.5). Здесь надо инициализировать ЭУ список окна, использовав соответствующий открытый атрибут класса DO2 – управляющую переменную m_L1, связанную с указанным ЭУ. Для этого в обработчике перед запуском ДО выполним соответствующие настройки

```
void CChildView::OnDialogsNotmodal( )
{     m_do2.ShowWindow( SW_SHOW );
      m_do2.m_L1.AddString("string 1");
      m_do2.m_L1.AddString("string 2");
      m_NotModalState = 1;   }  .
```

Задание 2. Создать приложение на основе типового каркаса mfc на базе диалогового окна (тип Dialog based). Требуется обеспечить:

1. Ввод строки и ее отображение с помощью окон редактирования (группа ЭУ Edits). Действия производятся нажатием кнопки Enter.

2. Выбор строки из списка (ЭУ List) с отображением выбранной строки в окне редактирования (ЭУ Edit). Выбор производится двойным щелчком левой клавиши мыши. Исходное заполнение списка задается при инициализации.

3*. При завершении работы приложения (сообщение WM_DESTROY) выводить комментирующее сообщение и значения введенных и выбранной строки.

4*. По нажатии левой клавиши мыши выводить сообщение с информацией, аналогичной пункту About диалогового окна.

5*. По нажатии правой клавиши мыши выводить окно, аналогичное окну About диалогового окна.

6*. Для работы с результатами ввода и выбора строк использовать пользовательский класс (его описание разместить: а) в файлах *.h, *.cpp описания самого диалогового окна; б) в дополнительных файлах *.h, *.cpp).

Примерный вид интерфейса представлен ниже (рисунок 120).

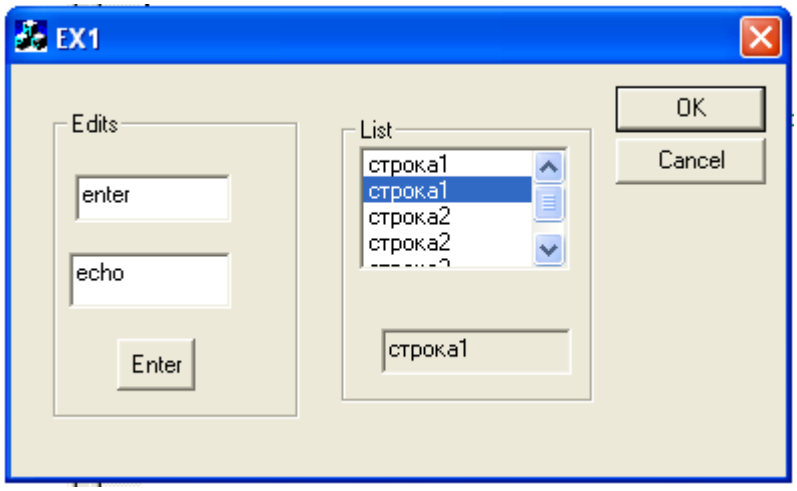


Рисунок 120 – Главное окно приложения

ЛАБОРАТОРНАЯ РАБОТА № 7, 8 “Разработка консольных приложений в C#”

ЦЕЛЬ РАБОТЫ. 1. Изучение базовых средств C# для работы в консольном режиме. 2. Разработка консольных приложений с использованием управляемых данных, безопасных кодов.

СОДЕРЖАНИЕ ОТЧЕТА. Описание реализованных приложений.

ЗАДАНИЯ ДЛЯ ВЫПОЛНЕНИЯ

Задание 1. Изучить теоретический материал (лекция 12, параграфы “Платформа MS.NET, Каркас консольного приложения и др.”, лекция 13, параграфы “Характеристика типов данных, Скалярные типы, преобразование типов, Организация вычислений”).

Задание 2. Выполнить задачи (лекция 13, параграф “Скалярные типы, преобразование типов, Организация вычислений”).

1. Изучите состав типового каркаса консольного приложения языка C#.

2. Воспроизведите код программы для ввода и преобразования данных скалярных типов (из раздела Скалярные типы, преобразование типов) и код программы, демонстрирующей использование типовых исключений для контроля ввода данных (из параграфа Организация вычислений).

Задание 3. Выполнить задачи (лекция 13, параграф “Консольный ввод-вывод”).

1. Повторите описание, инициализацию переменных базовых скалярных типов. Выведите данные разными способами. Используйте для преобразования данных разные методы, включая

```
Double.TryParse(s, out xmin);
Double.TryParse(s, out xmax);
n = System.Convert.ToInt16(s);  .
```

2. Организуйте ввод-вывод данных - результатов вычисления заданной функции. Приведите вывод результатов к табличной форме.

Задание 4. Выполнить задачи (лекция 14, параграф “Работа с данными символьного типа”).

1. Воспроизведите на языке C# описанные в указанном параграфе примеры (в синтаксисе CLI) и определите их назначение.

2. При записи в коде C# внесите изменения: - используйте тип char и оператор &&; - уберите приведение типов safe_cast; - в качестве типа int используйте типы Int16, Int32.

Задание 5. Выполнить задачи (лекция 14, параграф “Массивы”).

1. Воспроизведите на языке C# описанные в указанном параграфе примеры (в синтаксисе CLI) и определите их назначение.

2. Используйте приведенный ниже фрагмент для разработки приложения для работы с двумерным массивом (для ввода, вывода значений)

```
int nrows(4);
int ncols(5);
array<int, 2>^ values(gcnew array<int, 2>(nrows, ncols));
for(int i = 0 ; i < nrows ; i++)
   for(int j = 0 ; j < ncols ; j++)
      values[i,j] = (i+1)*(j+1);  .
```

3. Создайте приложение для поиска максимума в трехмерном массиве.

4. Удалите в заданном двумерном массиве строку, столбец с указанным номером.

5. Организуйте произведение, транспонирование матриц, используя соответствующие массивы.

6. Создайте приложение с функцией для сортировки числовых значений в одномерном массиве.

7. Создайте ступенчатый массив из двух одномерных массивов. Заполните массив, выведите по отдельности каждый из одномерных массивов. Выполните поиск максимального значения в массиве.

8. Создайте ступенчатый массив из двух двумерных массивов. Заполните массив, выведите значения, в т.ч. для каждого из двумерных массивов. Выполните поиск максимального значения в каждом из двумерных массивов. Суммировать элементы массива.

Задание 6. Выполнить задачи (лекция 14, параграф “Работа со строковыми данными”).

1. Опишите константный массив строк. Заполните массив данными (введите данные) и выведите строки. Выполните лексикографическую сортировку элементов массива.

2. Воспроизведите на языке C# описанные в указанном параграфе примеры (в синтаксисе CLI). Не используйте префикс L.

3. Создайте приложение с функцией для удаления из строки пробелов (для этого используйте метод String.Replace(”ЗаменяемыйСимвол”, ”ЗамещающийСимвол”).

Задание 7. Разработайте методы с указанными ниже назначением и прототипами (лекция 15, параграф “Особенности использования методов классов”). Реализуйте их, например, как static-методы пользовательского класса FUNCTIONS. Вызов функций осуществите в методе Main каркаса приложения.

1. Метод для обмена данных между переменными x, y

static public void TO_SWAP(ref int x, ref int y).

2. Метод для получения целой Whole и дробной Frac частей числа Number

static public void TO_GET_PARTS(double Number, out int Whole, out double Frac).

3. Метод для получения целой Whole и дробной Frac частей числа Number (исходное число обнулить)

static public void TO_GET_PARTS(ref double Number, out int Whole, out double Frac).

4. Метод с переменным числом параметров для вывода комментария Caption и поиска минимального значения среди значений Numbers. Проверить работу функции для разных вариантов вызова

static public int MIN_VALUE(string Caption, params int [ ] Numbers).

5. Метод для расчета числа положительных значений NumberOfPositive в массиве Numbers и возврата преобразованного массива Numbers (все члены умножить на два)

static public int[ ] MODIFIED_ARRAY(int [ ] Numbers, out int NumberOfPositive).

6. Создайте класс ABOUT_SWAP для хранения двух целых значений a, b, с конструктором с параметрами и методом TO_SHOW( ).

Добавьте метод для обмена значений двух объектов класса ABOUT_SWAP

public void TO_SWAP(ref ABOUT_SWAP rObject1, ref ABOUT_SWAP rObject2) .

Добавьте метод для создания нового объекта класса ABOUT_SWAP с заданными аргументами int i, int j и возврата соответствующей ссылки на этот объект

public ABOUT_SWAP ABOUT_SWAP_OBJECT(int i, int j) .

Добавьте деструктор ~ABOUT_SWAP( ) с выводом соответствующего сообщения и функцией ReadKey( ) для приостановки работы метода. Проанализируйте моменты запуска деструктора.

7. Метод для обмена значений двух объектов класса ABOUT_SWAP

static public void TO_SWAP(ref ABOUT_SWAP rObject1, ref ABOUT_SWAP rObject2) .

8. Метод для создания объекта класса ABOUT_SWAP с заданными аргументами int i, int j и возврата соответствующей ссылки

static public ABOUT_SWAP ABOUT_SWAP_OBJECT (int i, int j) .

Задание 8. Выполнить задачи (лекция 15, параграф “Структуры”).

1. Воспроизведите описанный в параграфе пример на языке C#, использовав для описания объектов соответствующую структуру. Атрибуты структуры сделайте открытыми.

2. Добавьте свойство для работы с атрибутом Feet – требуется возвращать текущее значение, а также поддерживать его установку как Feet = value с контролем положительности значения value. Проанализируйте результат присваивания друг другу ссылок на объекты типа структура.

Задание 9. Выполнить задачи (лекция 15, параграф “Классы. Свойства”).

1. Воспроизведите в языке C# описанный в параграфе пример (класс BOX). Используйте приведенный в разделе образец и добавьте к классу BOX два свойства: - public double prLength для возврата или установки значения Length с проверкой корректности

значения value; - public double prVolume для получения объема объекта как Length * Width * Height.

2. Создайте приложение для работы с классом (INT) для хранения и обработки целого значения. Предусмотрите все типы конструкторов.

3. Добавьте в предыдущем приложении свойство (prValue).

4. Создайте приложение для работы с классом (STR_LIST) для хранения и обработки списка (массива) строк.

Задание 10. Выполнить задачи (лекция 16, параграф “Перегрузка операторов (операций)”).

1. Воспроизведите описанный в параграфе пример на языке C#. В классе LENGTH используйте только один атрибут Feet. Добавьте в класс метод public int FEET( ).

2. Выполните перегрузку следующих операторов

public static LENGTH operator+ (LENGTH rObjec1, LENGTH rObjec2);
public static LENGTH operator/ (LENGTH rObjec1, LENGTH rObjec2);
public static LENGTH operator/ (LENGTH rObjec1, int Objec2);
public static int operator- (LENGTH rObjec1, LENGTH rObjec2);
public static LENGTH operator* (double Objec1, LENGTH rObjec2);
public static LENGTH operator++ (LENGTH rObject); .

3. Создайте приложение для работы с классом (INT) для хранения и обработки целого значения. Предусмотрите перегрузку оператора + (для разных вариантов сочетаний типов операндов) и оператора ++.

Задание 11. Выполнить задачи (лекция 16, параграф “Наследование классов”).

1. Воспроизведите приведенный в параграфе пример. Закомментируйте метод TO_SHOW_VOLUME() в производном классе и проанализируйте работу приложения.

2. При работе с объектами BOX продемонстрируйте использование динамического полиморфизма. Закомментируйте метод TO_SHOW_VOLUME( ) в базовом классе и проанализируйте работу приложения.

3. Создайте путем наследования приложение для работы с классом (TWO_INT) для хранения и обработки двух целых значений.

Задание 12. Выполнить задачи (лекция 16, параграф “Интерфейсы”).

1. Воспроизведите описанный в параграфе пример.

2. Создайте приложение для работы с классом INT (хранение и обработка целых значений) через интерфейс, обеспечивающий свойство Value для доступа к значению и метод TO_SHOW для вывода хранимого значения.

3. Создайте приложение для работы с двумя классами (один для хранения целого значения, другой для хранения строки) через общий интерфейс, предоставляющий методы установки значений TO_SET и их вывода TO_SHOW.

3 РАЗДЕЛ КОНТРОЛЯ ЗНАНИЙ

Перечень вопросов, выносимых на экзамен

1. Состав и порядок функционирования оконного приложения.
2. Сообщения, классификация, программная посылка сообщений.
3. Системные сообщения, обработка. Сообщения “мыши”, клавиатуры.
4. Командные и извещающие сообщения, обработка.
5. Перерисовка, программный вызов перерисовки.
6. Состав типового каркаса приложения (ТКП).
7. Обработчик сообщений главного окна ТКП.
8. Понятие класса окна, виды окон. Жизненный цикл окна.
9. Структура WNDCLASS. Описание и регистрация класса окна.
10. Создание и визуализация окон типа ”главное”. Функция CreateWindow.
11. Сообщения WM_SHOWWINDOW, WM_ACTIVATE, WM_CLOSE, WM_DESTROY и их обработка.
12. Управление контекстом устройства. Функции TextOut, MessageBox.
13. Вывод в клиентскую область окна с учетом межстрочного интервала, физической длины строки.
14. Меню, последовательность разработки, структура секции обработчика сообщений меню.
15. Диалоговые окна. Особенности использования клавиатуры, кнопок с IDOK, IDCancel.
16. Функции DialogBox, EndDialog и их использование.
17. Последовательность разработки приложений с диалоговым окном, в т.ч. в качестве главного.
18. Структура обработчика сообщений диалогового окна.
19. Пример активизации диалогового окна через пункт главного меню, ”мышью”.
20. Элементы управления (ЭУ). Извещающие сообщения. Функции для работы с ЭУ.
21. Элемент управления edit. Характеристика, настройка, управление.
22. Элемент управления listbox. Характеристика, настройка, управление.
23. Обработка события смены выбора строки в listbox как добавление к списку новой
24. MFC. Основные классы.
25. MFC. Классы, методы, наследуемые для создания каркаса приложения.
26. MFC. Классы, методы, для работы с окнами в приложении.
27. MFC. Диаграмма классов типового каркаса приложения (ТКП). Код типового каркаса приложения (ТКП).
28. MFC. Обработка сообщений. Макрокоманды, карты сообщений, обработчики.
29. MFC. Вывод в клиентскую область окна. Контекст устройства. Перерисовка. Методы.
30. MFC. Обработка сообщений “мыши” и нажатия клавиш клавиатуры.
31. MFC. Описание классов типового каркаса приложения.
32. MFC. Меню. Этапы разработки, подключение, обработка сообщений. Пример.
33. MFC. Понятие, этапы разработки диалоговых окон.
34. MFC. Типовые диалоговые окна и их классы. MessageBox. Примеры.
35. MFC. Класс CDialog, обработчики. Описание класса диалогового окна. Пример.

36. MFC. Сообщения диалогового окна, диаграмма состояний.

37. MFC. Создание объектов, активизация, инициализация и завершение работы диалогового окна. Немодальные окна.

38. MFC. Виды элементов управления, обработка сообщений ЭУ.

39. MFC. ЭУ кнопка, сообщения, средства управления, обработка сообщений.

40. MFC. ЭУ окно редактирования, сообщения, средства управления, обработка сообщений.

41. MFC. ЭУ список, сообщения, методы. Инициализация, чтение, добавление-удаление записей, обработка сообщений.

42. MFC. Потоковая многозадачность. Классы и методы для ее управления. Потоковая функция.

43. MFC. Особенности разработки меню и диалоговых окон с использованием средств автоматизации.

44. C#. MS NET-платформа, .NET фреймворк, CLR.

45. C#. Классификация типов. CTS. Базовые скалярные типы.

46. C#. Работа с ссылочными типами. Преобразование типов данных. Консольный ввод-вывод.

47. C#. Класс Object.

48. C#. Массивы: описание, инициализация, методы и свойства.

49. C#. Символы и строки: описание, инициализация, методы и свойства.

50. C#. Структуры. Описание и использование. Примеры.

51. C#. Классы. Описание и использование. Примеры.

52. C#. Методы. Описание и использование. Примеры.

53. C#. Свойства и индексаторы классов. Примеры.

54. C#. Перегрузка операторов. Примеры.

4 ВСПОМОГАТЕЛЬНЫЙ РАЗДЕЛ

Учебная программа

Р-1 2024

Учреждение образования
«Брестский государственный технический университет»

УТВЕРЖДАЮ
Проректор по учебной работе
_____ М.В. Нерода
28.06. 2024 г.

Регистрационный № УД-24-1-303/уч.

Проектирование программного обеспечения интеллектуальных систем

Учебная программа учреждения высшего образования по учебной дисциплине для специальности:
6-05-0611-03 Искусственный интеллект

2024

Учебная программа составлена на основе образовательного стандарта специальности ОСВО 6-05-0611-03-2023 и учебного плана специальности 6-05-0611-03 «Искусственный интеллект».

СОСТАВИТЕЛИ:
Муравьев Г.Л., доцент кафедры интеллектуальных информационных технологий, кандидат технических наук, доцент, старший научный сотрудник;
Мухов С.В., доцент кафедры интеллектуальных информационных технологий, кандидат технических наук, доцент

РЕЦЕНЗЕНТЫ:
Грицук Д.В., заведующий кафедрой прикладной математики и информатики Брестского государственного университета им. А.С. Пушкина, кандидат физико-математических наук, доцент;
Махнист Л.П., доцент кафедры математики и информатики учреждения образования «Брестский государственный технический университет», кандидат технических наук, доцент

РЕКОМЕНДОВАНА К УТВЕРЖДЕНИЮ:
Кафедрой интеллектуальных информационных технологий
Заведующий кафедрой В.А. Головко
(протокол № 9 от 17.06.2024)

Методической комиссией факультета электронно-информационных систем
Председатель методической комиссии С.С. Дереченник
(протокол № 9 от 22.06. 2024);

Научно-методическим советом БрГТУ (протокол № 5 от 28.06 2024)

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

Дисциплина «Проектирование программного обеспечения интеллектуальных систем» относится к модулю «Аппаратное и программное обеспечение интеллектуальных систем» компонента учреждения образования.

Дисциплина является одной из базовых в процессе подготовки студентов в области разработки программ, использования ИТ. Дисциплина носит системный характер, обеспечивает выработку систематизированных знаний по основам проектирования программ, имеет выраженную практическую направленность.

Цель преподавания учебной дисциплины:

изучение принципов, подходов, методов, средств и технологий проектирования, реализации и развития программ, программных систем.

Задачи учебной дисциплины:

- приобретение знаний о языках программирования разного уровня, основах парадигм программирования, средствах разработки интеллектуальных систем;
- изучение принципов построения программных систем на основе языков программирования разного уровня;
- формирование навыков использования методов и технологий проектирования и программирования систем с применением современных ИТ.

В результате изучения дисциплины формируются компетенции:

Проектировать программы в области интеллектуальных информационных систем с использованием современных языков программирования и средств разработки программ, применять навыки выбора парадигмы и языка программирования при решении конкретных задач (СК-8). В том числе

академические:

- владение исследовательскими навыками, умение работать самостоятельно;
- иметь навыки, связанные с использованием технических устройств, управлением информацией и работой с компьютером;
- владение основными методами, способами и средствами разработки программ на базе ИТ и компьютерной техники;

социально-личностные:

- умение работать в команде;

профессиональные:

- ставить и специфицировать реальные прикладные задачи с целью их решения с использованием компьютерной техники;
- проектировать компоненты и интерфейсы в рамках разрабатываемой программной или программно-технической системы.

В результате изучения учебной дисциплины студент должен:

знать:

- языки программирования разного уровня и назначения;
- методологические основы парадигм программирования;
- назначение, классификацию, состав инструментальных средств разработки программного обеспечения;

уметь:

- использовать методы и технологии проектирования и программирования программного обеспечения;

- разрабатывать программные проекты и их фрагменты в соответствии со стандартами и с применением современных методик и инструментальных средств;

владеть:

- базовыми методологиями проектирования программ;

- современными инструментальными средствами реализации программ и их фрагментов;

- навыками применения знаний для решения практических задач.

Изучение дисциплины базируется на знаниях, полученных студентами при изучении: дисциплины “Основы алгоритмизации и программирования”; - общематематических (блок “Математика”) и специальных дисциплин (Теоретико-множественные основы интеллектуальных систем); - специальных дисциплин в области компьютерных информационных технологий (Традиционные и интеллектуальные информационные технологии).

План учебной дисциплины для дневной формы получения высшего образования

Код специальности	Наименование специальности	Курс	Семестр	Всего учебных часов	Количество зачетных единиц	Аудиторных часов (в соответствии с учебным планом УВО)					Академических часов на курсовой проект (работу)	Форма промежуточной аттестации
						Всего	Лекции	Лабораторные занятия	Практические занятия	Семинары		
6-05-0611-03	Искусственный интеллект	2		**324**	**9**	144	64	64	16	-	-	
			3			80	32	32	16	-	-	зачет
			4			64	32	32	-	-	-	письменный экзамен

1. СОДЕРЖАНИЕ УЧЕБНОГО МАТЕРИАЛА

1.1. ЛЕКЦИОННЫЕ ЗАНЯТИЯ, ИХ СОДЕРЖАНИЕ

3 СЕМЕСТР

Раздел 1. КОНЦЕПТУАЛЬНЫЕ ОСНОВЫ ПРОЕКТИРОВАНИЯ ПРОГРАММ. ПРОГРАММНАЯ ИНЖЕНЕРИЯ

Тема 1. Введение. Предмет дисциплины

Предмет, цели, задачи дисциплины. Обзор изучаемых языков программирования, инструментальных средств, средств описания предметных областей и проектных решений, технологий разработки программ. Обзор литературных источников по темам учебной программы.

Тема 2. Основы проектирования

Программная система (ПС), сложность и классификация ПС. Системная, программная инженерия. Качество ПС, метрики качества ПС.

Понятие проектирования и технологии разработки ПС. Этапы разработки ПС, процессы. Документы, регламентирующие разработку. Понятие жизненный цикл (ЖЦ), модели ЖЦ. Управление разработкой ПС. Виды программной документации.

Парадигмы разработки программ. Объектная парадигма. Понятие объектной модели: принципы, виды моделей, средства разработки и описания. Особенности разработки ПС в объектно-ориентированной парадигме.

Раздел 2. КОНСТРУИРОВАНИЕ ПРОГРАММ (ПОДХОДЫ, СРЕДСТВА)

Тема 1. Объектно-ориентированное программирование средствами языка C++

Характеристика, особенности языка программирования C++. Ссылочный тип: сравнение с указателями, описание, инициализация, использование. Управление “кучей”. Особенности работы с функциями: спецификатор void, прототипы функций, аргументы по умолчанию. Полиморфизм как перегрузка функций.

Классы, члены классов (свойства, операции, методы), объекты классов, описание на UML. Указатель this. Спецификаторы, ситуации доступа (видимость членов классов). Использование объектов в функциях, операциях.

Инициализация объектов, конструкторы по умолчанию, с параметрами, конструкторы копирования, перегрузка конструкторов. Деструкторы. Друзья классов: назначение, синтаксис, особенности использования. Иллюстрация отношения зависимости (типа использование) на примере друзей классов, описание на UML.

Принципы и особенности перегрузки операторов классов на базе операций (функций) классов и функций-друзей.

Особенности описания и использования классов со структурными и динамическими атрибутами. Статичные члены классов, особенности описания (в том числе на UML) и использования.

Наследование классов как реализация принципа иерархичности, прямое одиночное наследование базовых классов, описание на UML. Описание производных классов, особенности описания операций, конструкторов, деструкторов. Модификаторы и ситуации доступа (видимость членов классов). Понятие множественного наследования, виртуальных и чисто виртуальных операций (функций) классов, динамического полиморфизма, полиморфных, абстрактных классов.

Пространство имен. Отношения ассоциации (композиция и агрегация) в иерархиях классов, описание на UML, использование.

Программы, устойчивые к ошибкам. Исключения, идентификация ошибок, синтаксис, особенности использования команд try, catch, throw. Библиотечные исключения.

Понятие порождающей парадигмы, примеры реализации. Мета-программирование, макросы.

Шаблоны как родовые функции и классы, описание, особенности использования. Отношение зависимости типа связывание, описание на UML.

Библиотека STL, обзор базовых элементов (контейнеры, аллокаторы, итераторы, алгоритмы и др.).

Виды контейнеров STL, особенности использования.

Объектно-ориентированная библиотека ввода-вывода visual C++. Иерархия шаблонов классов ввода-вывода. Форматируемый, не форматируемый, файловый ввод-вывод. Управление вводом-выводом, флаги, манипуляторы. Перегрузка операторов ввода-вывода.

4 СЕМЕСТР

Раздел 2. КОНСТРУИРОВАНИЕ ПРОГРАММ (ПОДХОДЫ, СРЕДСТВА)

Тема 2. Создание оконных приложений средствами языка C++

Понятие аппаратная, программная платформа, виртуальная машина. Характеристика операционной системы как виртуальной машины (на примере ОС windows). Особенности именования, типы, структуры данных ОС windows.

Приложения, управляемые сообщениями, на примере оконных windows-приложений. Состав, структура приложений (графический интерфейс пользователя, графические ресурсы, иерархии окон, каркас приложения, обработчики сообщений).

Диспетчирование сообщений. Порядок функционирования приложения. Описание оконных приложений диаграммами UML - диаграммы прецедентов и состояний (назначение, элементы, отношения), Примеры использования.

Сообщения, параметры сообщений, классификация. Особенности командных, извещающих сообщений. Принципы обработки сообщений. Обработчики.

Типовой каркас оконного windows-приложения (ТКП): характеристика модулей Главное окно, Обработчик главного окна, функций. Схемы иерархии функций модулей, порядок работы. Код ТКП. Описание приложений на базе ТКП на UML.

Классы (стили) окон. Последовательность работы с окнами. Окно типа “главное”: состав, порядок описания и регистрации, структура WNDCLASS, оконные функции. Масштабируемые окна. Понятие перерисовки, сообщение и ситуации перерисовки, программный вызов перерисовки. Клиентская область окна (КО), контекст устройства (КУ), вывод данных в клиентскую область. Особенности вывода текстовых данных.

Графические ресурсы, виды, средства, порядок создания. Разработка, использование меню окон типа “главное”. Диалоговые окна (ДО): виды, особенности работы и использования. Окна сообщений, специализированные окна. Элементы управления (ЭУ): виды, классификация, описание базовых ЭУ. api-функции

управления ЭУ. Порядок разработки и использования ДО. Описание приложений с ДО на UML. Программное управление сообщениями.

Тема 3. Создание mfc-приложений

Библиотека MFC: общая характеристика. Классы, наследуемые от CObject, автономные классы, глобальные функции. Классы, используемые для создания каркасов приложений, графических интерфейсов. Типовой каркас mfc-приложения (ТКП): состав классов, базовых членов, операций классов, порядок функционирования. Описание ТКП, приложений на базе ТКП на UML. Прототипы базовых операций классов ТКП (InitInstance, Create, ShowWindow и др.). Пример создания окна типа “главное”. Управление окнами. Код ТКП.

Обработка сообщений: виды сообщений, карты сообщений, макрокоманды включения сообщений. Обработчики сообщений, встроенные обработчики, прототипы типовых обработчиков. Клиентская область: перерисовка, управление контекстом устройства, вывод данных, работа с текстом.

Разработка интерфейсов на базе библиотеки MFC. Окна сообщений, специализированные ДО. Пользовательские диалоговые окна, класс CDialog, обработчики, использование кнопок IDOK, IDCANCEL. Использование ДО в качестве главного окна, в составе окон. Описание на UML. Особенности использования базовых ЭУ.

Управление пользовательскими меню, динамические меню. Использование мастеров автоматизации разработки приложений (автокаркасы, управление ЭУ, каналы DDX и т.п.). Управление многозадачностью, потоковые функции.

Тема 4. Создание приложений средствами C#

Платформа MS.NET, .NET Framework, CLR. Характеристика языка программирования C#. Сборка, решение, манифест. Библиотека System, пространства имен. Каркас консольного приложения.

Типы данных .NET, типы языка C#. Скалярные типы, преобразование типов. Организация вычислений. Консольный ввод-вывод. Структуры.

Ссылочные типы. Массивы, строки. Классы, свойства, индексаторы.

Перегрузка операций. Интерфейсы. Наследование. Библиотека ввода-вывода.

1.2. ПЕРЕЧЕНЬ ТЕМ ЛАБОРАТОРНЫХ ЗАНЯТИЙ, ИХ НАЗВАНИЕ

3 СЕМЕСТР

1. Классы и объекты. Инициализация объектов.
2. Классы со структурными и динамическими атрибутами.
3. Друзья классов. Отношение зависимости между классами. Перегрузка операторов.
4. Наследование классов.
5. Отношения агрегации и композиции между классами. Пространства имен.
6. Исключения. Пользовательские шаблоны функций и классов.

7. Шаблоны STL.

8. Шаблоны STL. Организация ввода-вывода.

4 СЕМЕСТР

1. Каркасы оконных windows-приложений. Обработка сообщений. Работа с клиентской областью окна.

2. Создание интерфейсов windows-приложений. Работа с окнами, диалоговыми окнами.

3. Создание интерфейсов windows-приложений. Работа с меню, элементами управления.

4. Каркасы mfc-приложений. Обработка сообщений. Работа с клиентской областью окна. Стандартные мастера.

5. Создание интерфейсов mfc-приложений. Работа с окнами, меню, элементами управления.

6. Средства автоматизации разработки mfc-приложений. Каркас документ-вид.

7. Каркас консольного приложения на языке C#. Типы данных. Преобразование типов. Консольный ввод-вывод.

8. Классы. Методы. Перегрузка операций. Наследование.

1.3. ПЕРЕЧЕНЬ ТЕМ ПРАКТИЧЕСКИХ ЗАНЯТИЙ, ИХ НАЗВАНИЕ

3 СЕМЕСТР

1. Этапы разработки программ. Документирование процессов разработки. ГОСТ 19.701 Схемы алгоритмов, программ, данных и систем.

2. Техническое задание. Программа и методика испытаний. Описание программы. Описание применения.

3. Разработка классов. Описание классов и объектов на языке UML.

4. Друзья классов. Отношение зависимости между классами. Перегрузка операторов.

5. Наследование классов. Описание в UML.

6. Отношения агрегации и композиции классов.

7. Пользовательские шаблоны функций и классов. Шаблоны STL.

8. Шаблоны STL.

9. Организация ввода-вывода. Файловый ввод-вывод.

2. УЧЕБНО-МЕТОДИЧЕСКАЯ КАРТА УЧЕБНОЙ ДИСЦИПЛИНЫ
для дневной формы получения высшего образования

Номер раздела темы, занятия	Название раздела, темы	Количество аудиторных часов				количество часов самостоятельной работы	форма контроля знаний
		лекции	практические	лаборат. занятия	иное		
1	2	3	4	5	6	7	8
3 СЕМЕСТР							
	Раздел 1. КОНЦЕПТУАЛЬНЫЕ ОСНОВЫ ПРОЕКТИРОВАНИЯ						
1.1.1	Тема 1. Введение. Предмет дисциплины	2				4	
	Тема 2. Основы проектирования						
1.2.2	Программная система (ПС), сложность и классификация ПС	2	2			4	
1.2.3	Понятие проектирования и технологии разработки ПС	2	2			5	
	Лаб. работа № 1. Классы, объекты			4		4	защита отчета
1.2.4	Парадигмы разработки программ	2				4	
	Лаб. работа № 2. Классы со структурными и динамическими атрибутами			4		4	защита отчета
	Раздел 2. КОНСТРУИРОВАНИЕ ПРОГРАММ						
	Тема 1. Объектно-ориентированное программирование средствами языка C++						
2.1.5	Характеристика, особенности языка	2				4	
2.1.6	Классы, члены классов	2	2			4	
	Лаб. работа № 3. Друзья классов. Перегрузка операторов			4		4	защита отчета
2.1.7	Инициализация объектов. Друзья классов	2				4	контрольная работа
2.1.8	Перегрузка операторов	2	2			4	
	Лаб. работа № 4. Наследование классов			4		4	защита отчета
2.1.9	Классы со структурными и динамическими атрибутами	2				3	устный опрос
2.1.10	Наследование классов	2	2			4	
	Лаб. работа № 5. Отношения агрегации и композиции е			4		4	защита отчета
2.1.11	Пространство имен. Отношения ассоциации	2	2			4	
2.1.12	Программы, устойчивые к ошибкам	2				3	контрольная работа
	Лаб. работа № 6. Исключения. Пользовательские шаблоны функций и классов			4		4	защита отчета
2.1.13	Порождающая парадигма. Директивы условной компиляции	2				3	
	Лаб. работа № 7. Шаблоны STL			4		4	защита отчета
2.1.14	Шаблоны функций и классов	2	2			4	
2.1.15	Библиотека STL	2	2			4	
	Лаб. работа № 8. Шаблоны STL. Организация ввода-вывода			4		4	защита отчета
2.1.16	Организация ввода-вывода	2				4	
4 СЕМЕСТР							
	Раздел 2. КОНСТРУИРОВАНИЕ ПРОГРАММ						
	Тема 2. Создание оконных приложений средствами языка C++						

2.2.1	Аппаратная, программная платформа, виртуальная машина	2				4	
2.2.2	Приложения, управляемые сообщениями	2				4	
2.2.3	Порядок функционирования приложения	2				4	
2.2.4	Сообщения, параметры сообщений, классификация	2				4	
	Лаб. работа № 1. Каркасы оконных windows-приложений. Обработка сообщений			4		4	защита отчета
2.2.5	Типовой каркас оконного windows-приложения	2				4	устный опрос
2.2.6	Классы (стили) окон. Основы работы с окнами	2				4	
	Лаб. работа № 2. Создание интерфейсов windows-приложений			4		4	защита отчета
2.2.7	Графические ресурсы. Диалоговые окна, меню, ЭУ	2				3	
	Лаб. работа № 3. Создание интерфейсов windows-приложений			4		4	защита отчета
	Тема 3. Создание mfc-приложений						
2.3.8	Библиотека mfc. Типовой каркас mfc-приложения	2				3	
	Лаб. работа № 4. Каркасы mfc-приложений. Обработка сообщений			4		4	защита отчета
2.3.9	Обработка сообщений. Работа с клиентской областью окна	2				3	
2.3.10	Разработка интерфейсов на базе библиотеки MFC	2				3	устный опрос
	Лаб. работа № 5. Создание интерфейсов mfc-приложений			4		4	защита отчета
2.3.11	Управление пользовательскими меню. Многозадачность	2				3	
	Лаб. работа № 6. Средства автоматизации разработки mfc-приложений			4		4	защита отчета
	Тема 4. Создание приложений средствами C#						
2.4.12	Платформа MS.NET, .NET Framework, CLR	2				3	
2.4.13	Типы данных .NET, типы языка C#	2				3	устный опрос
2.4.14	Ссылочные типы. Классы, свойства, индексаторы	2				3	
	Лаб. работа № 7. Разработка консольных приложений на C#. Преобразование типов. Консольный ввод-вывод.			4		4	защита отчета
2.4.15	Перегрузка операций	2				3	
2.4.16	Интерфейсы. Наследование	2				3	
	Лаб. работа № 8. Разработка консольных приложений на C#. Классы. Перегрузка операций			4		4	защита отчета

3. ИНФОРМАЦИОННО-МЕТОДИЧЕСКАЯ ЧАСТЬ

3.1. Перечень литературы

Основная

1. Орлов, С.А. Программная инженерия: технологии разработки программного обеспечения: учебник для вузов / С.А. Орлов. – СПб.: Питер, 2018. – 640 с.

2. Павловская Т.А. C/C++. Программирование на языке высокого уровня: Учебник для вузов / Т.А. Павловская. – СПб.: Питер, 2021. – 461 с.

3. Общие сведения о программировании на C++ в Windows. Visual Studio 2015-2022 [Электронный ресурс]. – 2022. - Режим доступа: https://docs.microsoft.com/ru-ru/cpp/windows/overview-of-windows-programming-in-cpp?view=msvc-170. – Дата доступа: 27.05.2023.

4. Приложения MFC для рабочего стола. Visual Studio 2015-2022 [Электронный ресурс]. – 2022. - Режим доступа: https://docs.microsoft.com/ru-ru/cpp/mfc/mfc-desktop-applications?view=msvc-170. – Дата доступа: 27.05.2024.

5. Сидорина Т. Л. Самоучитель Microsoft Visual Studio C++ и MFC. - СПб.: БХВ-Петербург, 2009. — 848 с.
6. Павловская, Т. А. Программирование на языке высокого уровня C# / Т.А. Павловская – М. : Национальный Открытый Университет "ИНТУИТ", 2016. – 464 с.
7. Хултен Дж. Разработка интеллектуальных систем / пер. с анг. В. С. Яценкова. – М.: ДМК Пресс, 2019. – 284 с.

Дополнительная

8. Буч, Г. UML / Г. Буч, А. Якобсон, Дж. Рамбо. - СПб.: Питер, 2006. - 736 с.
9. Г. Буч. Объектно-ориентированный анализ и проектирование с примерами приложений на C++, 2-е изд./ Пер. с англ. – М.: Издательство БИНОМ, СПб: Невский диалект, 1998 г. – 560 с.
10. Одинцов И. Профессиональное программирование. Системный подход. – СПб.: БХВ-Петербург, 2002. – 512 с.
11. Финогенов К.Г. Win32. Основы программирования. М.: ДИАЛОГ-МИФИ, 2006. – 416 с
12. Шилдт Г. Самоучитель C++, 3-е изд. – СПб.: БХВ-Петербург, 2003. – 688 с.
13. Шилдт Г. MFC: Основы программирования. – М.: Изд. Группа BHV, 1997. – 560 с
14. Шилдт Г. C# 4.0: полное руководство. – М.: ООО И.Д. Вильямс, 2011. – 1056 с.
15. Рассел, С. Искусственный интеллект: современный подход/ С. Рассел, П. Норвиг; пер. с англ. - 2-е изд. - М. : Изд. дом «Вильямс», 2006. - 1408 с.

3.2. Перечень компьютерных программ, методических указаний и материалов для выполнения лабораторных работ

1. Среда программирования MS Visual Studio. САПР Rational Rose (или аналогичная). Средства построения UML-диаграмм.
2. Муравьев Г.Л. Методическое пособие “Основы создания windows-приложений в системе MS Visual Studio C++. Процедурный стиль”. – Брест.: БрГТУ, 2007. Часть 1. - 48 с.
3. Г.Л. Муравьев, В.И. Хвещук, В.Ю. Савицкий. Методическое пособие “Основы создания windows-приложений в системе MS Visual Studio C++. Процедурный стиль”. – Брест: БрГТУ, 2008. Часть 2. - 47 с.
4. Г.Л. Муравьев, В.И. Хвещук, В.Ю. Савицкий, С.В. Мухов. Методическое пособие “Основы создания windows-приложений в системе MS Visual Studio C++ на базе библиотеки MFC”. – Брест: БрГТУ, 2008. Часть 1. - 45 с.
5. Г.Л. Муравьев, В.И. Хвещук, В.Ю. Савицкий, С.В. Мухов. Методическое пособие “Основы создания windows-приложений в системе MS Visual Studio C++ на базе библиотеки MFC”. – Брест: БрГТУ, 2009. Часть 2. - 45 с.

6. Муравьев Г.Л., Мухов С. В., Шуть В.Н. Методическое пособие “Основы создания Windows-приложений в системе Microsoft Visual Studio C++ на базе библиотеки MFC. Использование элементов управления. Меню”, часть 3. – Брест: БрГТУ, 2010. – 48 с.

7. Муравьев Г.Л., Хвещук В.И., Мухов С.В. Методические указания к лабораторным работам “Разработка приложений на базе каркаса документ-вид”. – Брест: БрГТУ, 2011. – 46 с.

8. Муравьев Г.Л., Хвещук В.И., Мухов С.В. Методическое пособие “Разработка приложений на базе каркаса документ-вид (мастер MFC AppWizard)”. – Брест: БрГТУ, 2012. – 75 с.

9. Муравьев Г.Л. Мухов С.В., Хвещук В.И. Методическое пособие “Средства автоматизации разработки MFC-приложений” для студентов специальности ИИ. – Брест: БрГТУ, 2014. - 52 с. (электронный вариант).

10. Муравьев Г.Л., Мухов С.В. Методическое пособие “Работа с источниками данных средствами MS Visual Studio” для студентов специальности ИИ. – Брест: БрГТУ, 2015. - 73 с. (электронный вариант).

11. Муравьев Г.Л., Мухов С.В. Методические указания к выполнению лабораторных работ “ Разработка приложений с формами средствами visual C#” для студентов специальности ИИ. – Брест: БрГТУ, 2023. - 31 с.

3.3. Перечень средств диагностики результатов учебной деятельности

- письменные отчеты по лабораторным работам;
- устный опрос (зачет), письменный экзамен.

Текущая аттестация проводится в виде:

- защиты лабораторных работ на лабораторных занятиях;
- контрольного опроса по теоретической части курса.

При расчете итоговой отметки текущей аттестации учитываются вышеуказанные формы отчетности. Весовой коэффициент текущей аттестации $K_{тек}$ рассчитывается следующим образом $K_{тек} = (1 - K_{промеж})/M$, где $K_{промеж} = 0,4$ — весовой коэффициент промежуточной аттестации; M — число предусмотренных текущих аттестаций в семестре.

Результаты текущей аттестации учитываются при проведении промежуточной аттестации по дисциплине.

3.4. Методические рекомендации по организации и выполнению самостоятельной работы

Самостоятельная работа включает: - изучение лекционного материала (ведение конспекта лекций, проработку соответствующих разделов рекомендованной литературы по темам изучаемой дисциплины); - подготовку к выполнению лабораторных работ, что включает изучение необходимого теоретического материала, соответствующих методических указаний (см. методические указания, список которых приведен в разделе 3.2 учебной программы); - подготовку письменных отчетов по результатам выполнения лабораторных работ, контрольных заданий, подготовку к защите результатов лабораторных работ; - подготовку к зачету, письменному экзамену по дисциплине.

Ссылки на рекомендуемые источники, учебную литературу (в разрезе тем изучаемой дисциплины) представлены в таблице ниже.

Тема учебной дисциплины	Литература
Раздел 1. Концептуальные основы проектирования программ	
Тема 1. Предмет дисциплины	[1, с. 13-19, с. 20-23]
Тема 2. Основы проектирования	[1, с. 23-35, с. 48-50, с. 54-55, с. 178-279, с. 186], [2, с. 102-114]
Раздел 2. Конструирование программ	
Тема 1. Объектно-ориентированное программирование средствами языка C++	[2, с. 178-230, с. 265-284, с. 295-316, с. 343-347]
Тема 2. Создание оконных приложений средствами языка C++	[3], методические пособия и указания [2, 3]
Тема 3. Создание mfc-приложений	[5, с. 5-84, с.155-182, с. 217-227, с. 237-290], [4], методические пособия и указания [4-9]
Тема 4. Создание приложений средствами языка C#	[6, с. 4-7, с. 10-16, с. 27-31, с. 53-60, с. 87-114, с. 161-216], методические пособия и указания [11]