Гусин А.Н.

Методическое пособие

по работе в среде программирования
Microsoft Visual C++

Библиотека MFC

Программирование под Windows

Предисловие

В данном методическом пособии рассматриваются основные возможности среды программирования *Microsoft Visual C++*, описывается интерфейс, порядок создания приложений с использованием данного продукта. Кроме теоретической части, приводятся наглядные примеры, для более лучшего усвоения материала.

Кроме этого, в отдельном разделе, рассматривается создание приложений под *ОС Windows*, с необходимыми пояснениями, теоретической частью и примерами программ.

Также после каждого раздела существует список вопросов для самопроверки.

Введение

Система программирования *Visual C++* – один из наиболее полных и совершенных продуктов, предназначенных для разработки программного обеспечения.

Это высокоскоростная и удобная для программирования система, предлагающая широкий набор разнообразных инструментов проектирования для любого стиля программирования. В *VC++* функции системы существенно расширены. Новые компоненты содержат средства для программирования приложений (их построения и отладки), улучшенную реализацию технологий *ActiveX* и *Internet*, дополнительные возможности разработки баз данных, а также новые архитектуры приложений.

I. Обзор среды программирования MS Visual C++

Компилятор *MS Visual C++* содержит множество интегрированных средств визуального программирования. Ниже перечислены основные утилиты, которые можно использовать непосредственно из *Visual C++:*

- **Интегрированный отладчик**

  Разработчики компании *Microsoft* встроили первоначальный отладчик *Code View* непосредственно в среду *Visual C++.* Команды отладки вызываются из меню *Debug*. Встроенный отладчик позволяет пошагово выполнять программу, просматривать и изменять значения переменных и многое другое.

- **Встроенные редакторы ресурсов**

  Редакторы ресурсов позволяют создавать и модифицировать ресурсы Windows, такие как растровые изображения, указатели мыши, значки, меню, диалоговые окна и т.д.

- **Редактор диалоговых окон**

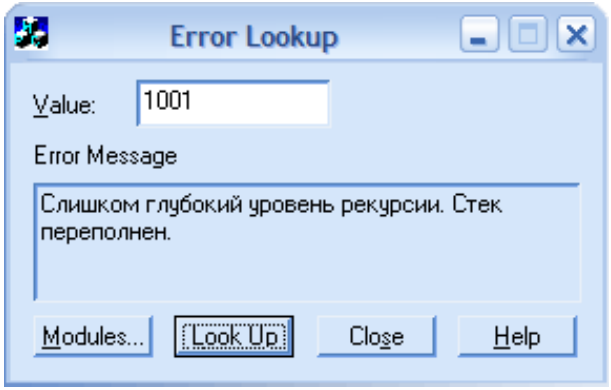
  Редактор диалоговых окон — это достаточно удобное средство, позволяющее легко и быстро создавать сложные диалоговые окна. С помощью этого редактора в разрабатываемое диалоговое окно можно включить любые элементы управления (например, надписи, кнопки, флажки, списки и т.д.).

- **Редактор изображений**

  Этот редактор позволяет быстро создавать и редактировать собственные растровые изображения, значки и указатели мыши. Пользовательские ресурсы данного типа сохраняются в файле с расширением RC и включаются в файлы сценариев ресурсов.

- **Error Lookup**

  Эта утилита позволяет просматривать и анализировать всевозможные сообщения об ошибках.

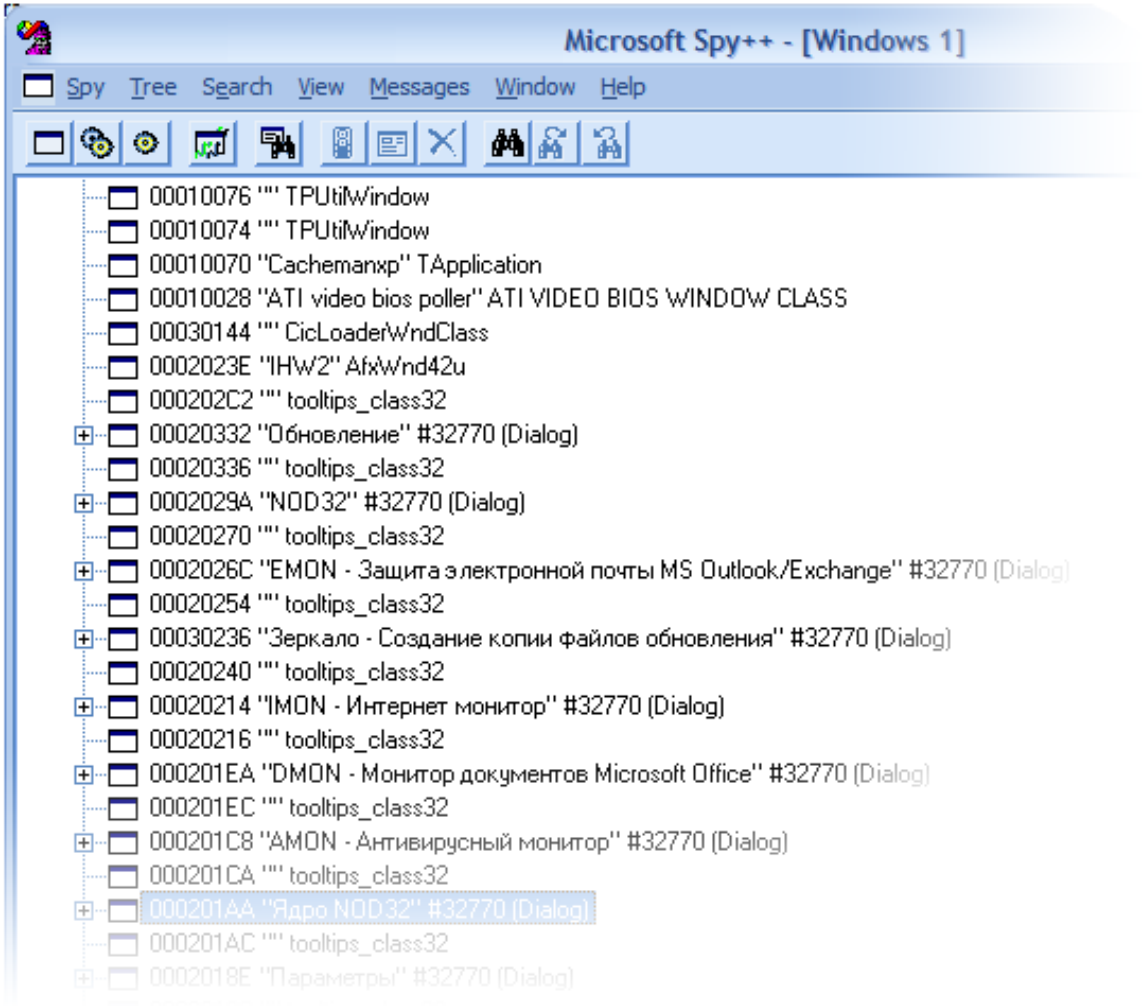


- **Stress Utility**

  Эта утилита позволяет захватывать системные ресурсы и используется для тестирования системы в ситуациях, связанных с недостатком системных ресурсов. В число захватываемых ресурсов входят глобальная и пользовательская динамические области (кучи), динамическая область *GDI*, свободные области дисков и дескрипторы файлов. Кроме того, утилита способна вести журнал событий, что помогает обнаруживать и воспроизводить аварийные ситуации в работе программы.

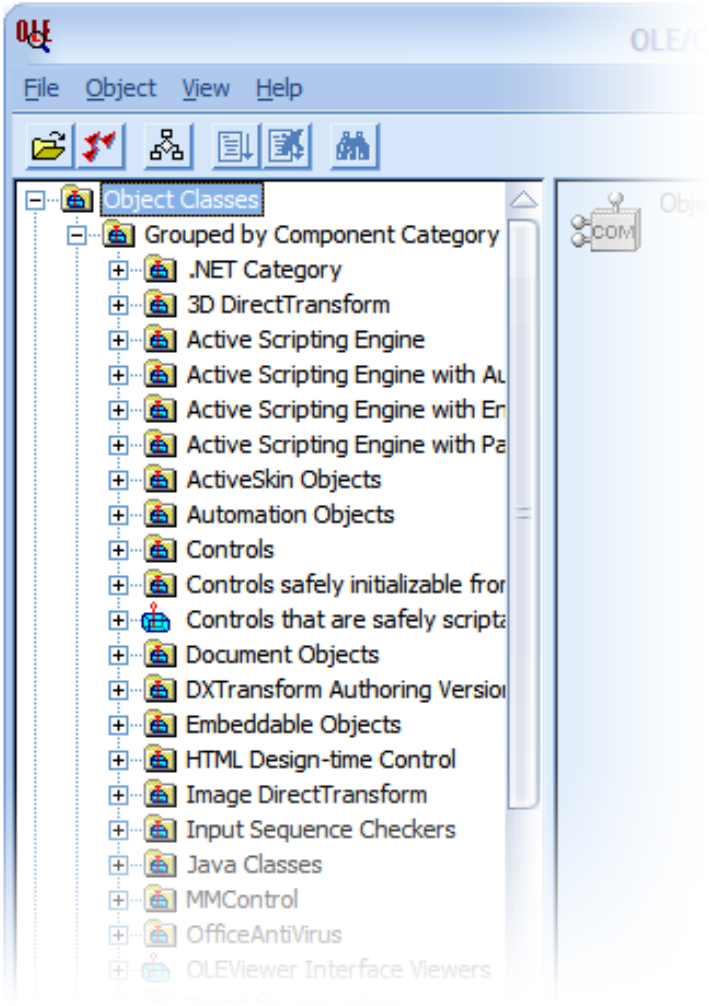
- **Spy++**

Эта утилита выводит сведения о выполняющихся процессах, потоках, существующих окнах и оконных сообщениях.



- **OLE Client/Server, Tools и View**

Утилита *OLE Viewer* отображает информацию об объектах *ActiveX* и *OLE*, инсталлированных на компьютере. Эта утилита также позволяет редактировать реестр и просматривать библиотеки типов. Утилиты *OLE Client* и *OLE Server* предназначены нтов и серверов.



Возможности компилятора

Компилятор *Visual C++* содержит много новых инструментальных средств и улучшенных возможностей, например:

- **Средства автоматизации и макросы**

  С помощью сценариев *Visual Basic* возможно автоматизировать выполнение рутинных и повторяющихся задач. *Visual C++* позволяет записывать в макрокомандах самые разные операции со своими компонентами, включая открытие, редактирование и закрытие документов, изменение размеров окон и т.д.

Настраиваемые панели инструментов и меню

В *Visual C++* стало легче настраивать панели инструментов и меню в соответствии с предпочтениями пользователя.

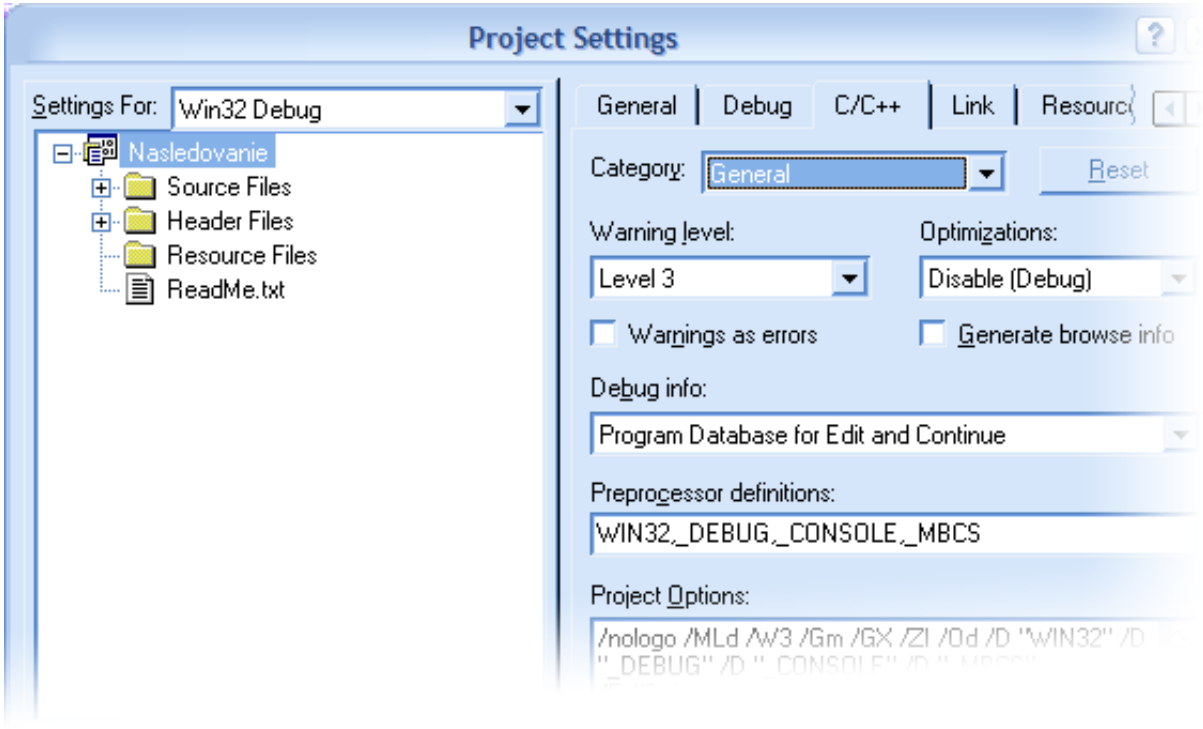
В частности, можно выполнять следующие действия:

- добавлять меню в панель инструментов;
- добавлять и удалять команды меню и кнопки панели инструментов;
- заменять кнопки панели инструментов соответствующими командами меню;
- создавать копии команд меню или кнопок панелей инструментов на разных панелях, с тем чтобы облегчить доступ к ним в разных ситуациях;
- создавать новые панели инструментов и меню;
- настраивать внешний вид существующих панелей инструментов и меню;
- назначать команды меню новым кнопкам панелей инструментов.

Опции компиляции

Компилятор *MS Visual C++* предоставляет огромные возможности в плане оптимизации приложений, в результате чего можно получить выигрыш как в отношении размера программы, так и в отношении скорости ее выполнения, независимо от того, что представляет собой приложение.

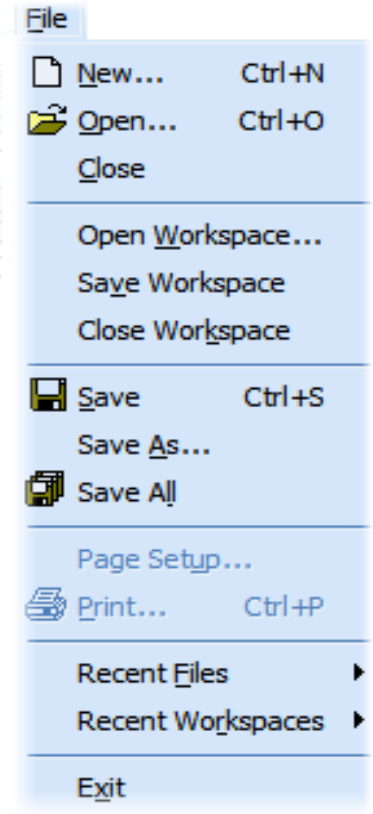
Опции компиляции позволяют оптимизировать программный код, сокращая его размер, время выполнения и время компиляции. Чтобы получить доступ к этим опциям, нужно в меню *Project* выбрать команду *Settings*.



Описание интерфейса среды *Visual C++*

- **Меню File**

В *Visual C++* в меню *File* собран стандартный для многих приложений *Windows* набор команд, предназначенных для манипулирования файлами:



1. *New* – по команде *New* открывается окно для выбора типа создаваемого файла, проекта или рабочего пространства
2. *Open* – команда открытия файла
3. *Close* – команда закрытия файла, причем закрывается акт т.е. текущее окно
4. *Open Workspace* – открытие рабочего пространства
5. *Save Workspace* – сохранение рабочего пространства
6. *Close Workspace* – закрытие рабочего пространства
7. *Save* – сохранение содержимого текущего окна

8. *Save As* – сохранение содержимого текущего окна в файле под новым именем
9. *Save All* – сохранение всех открытых файлов проекта (*.h, *.cpp)
10. *Recent Files* – список недавно открывавшихся файлов
11. *Exit* – выход из VC++
12. *Recent Workspaces* – список недавно открывавшихся проектов

- **Описание рабочего пространства среды Visual C++**

Под *рабочим пространством* понимается вся совокупность информации, хранимая в указываемом для *Workspace* каталоге. Эта информация хранится в нескольких служебных файлах.

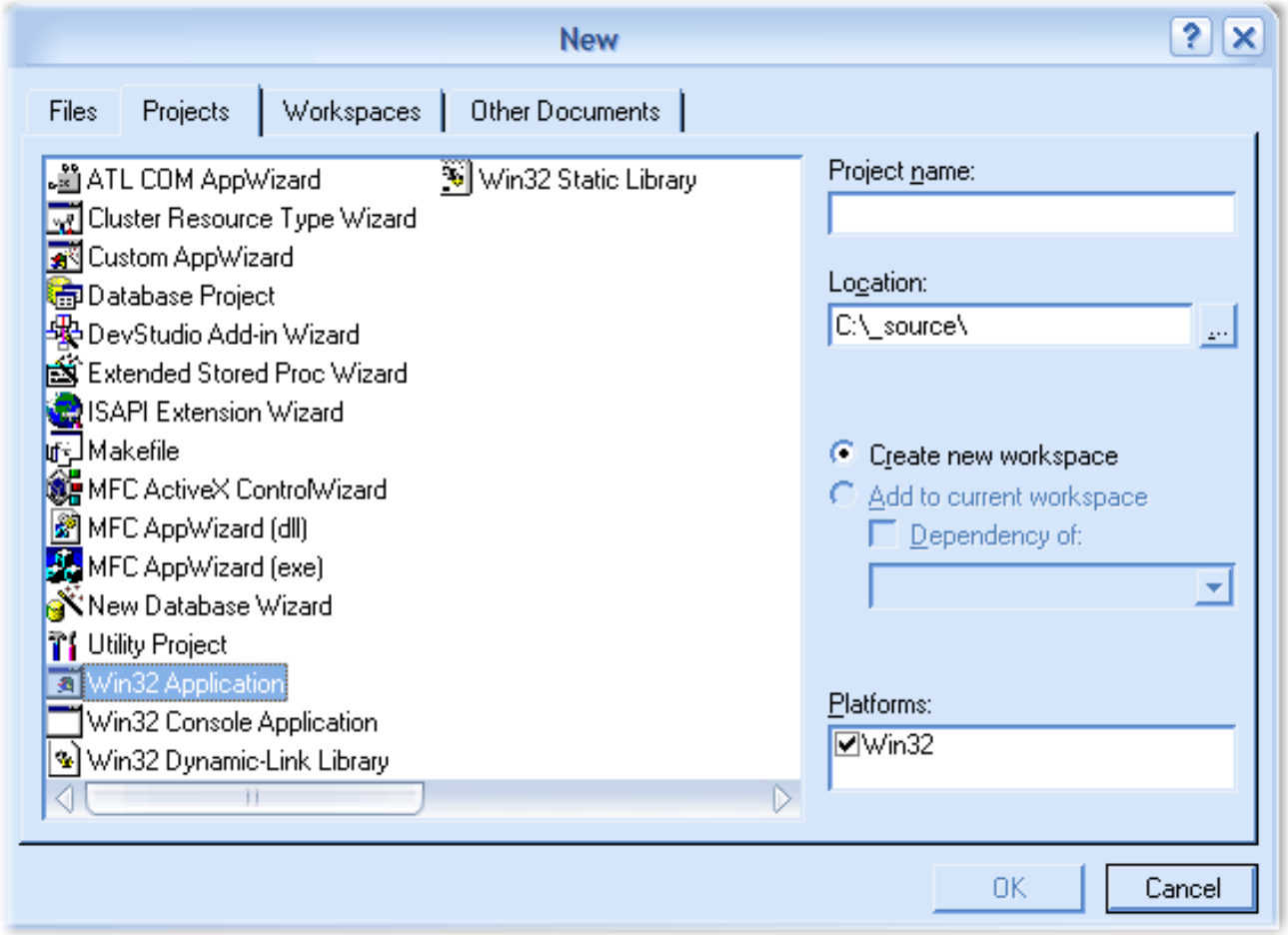
Рабочее пространство может быть пустым или содержать один или несколько проектов, причем это могут быть проекты различных типов. *Проектом* называется совокупность файлов, содержащих информацию об установках, конфигурации, ресурсах проекта, а также файлов исходного кода и заголовочных файлов. По умолчанию каждый проект помещается в отдельный каталог, одноименный с именем проекта.

При создании проекта имеется возможность как автоматического создания нового рабочего пространства, описывающего весь этот проект, так и добавления всех создаваемых файлов проекта в текущее рабочее пространство. Также имеется возможность создания пустого рабочего пространства.

Если рабочее пространство создается отдельно от проекта, то соответствующие служебные файлы по умолчанию помещаются в отдельный каталог. Рабочее пространство может содержать несколько проектов, как правило каждый в своем каталоге.

Создание нового проекта

Начинать разработку нового приложения лучше всего сразу с создания проекта, содержащего всю информацию о файлах проекта, ресурсах и классах. Для этого нужно выбрать пункт меню *File -> New* и в открывшемся диалоговом окне выбрать вкладку *Project*. На этой вкладке можно выбрать тип создава

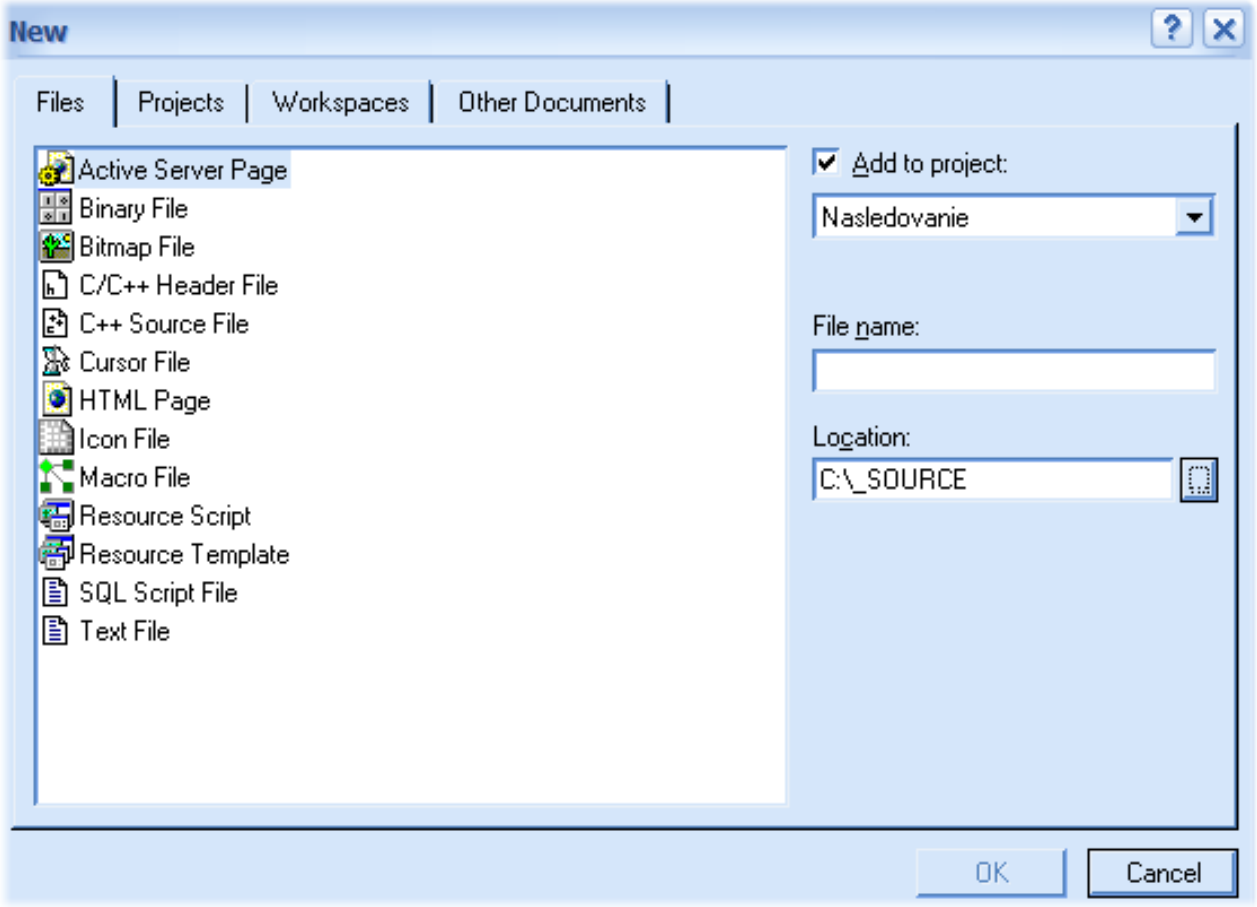


В поле *Project Name* указывается имя создаваемого проекта, а в поле *Location* путь по которому проект будет сохранен.

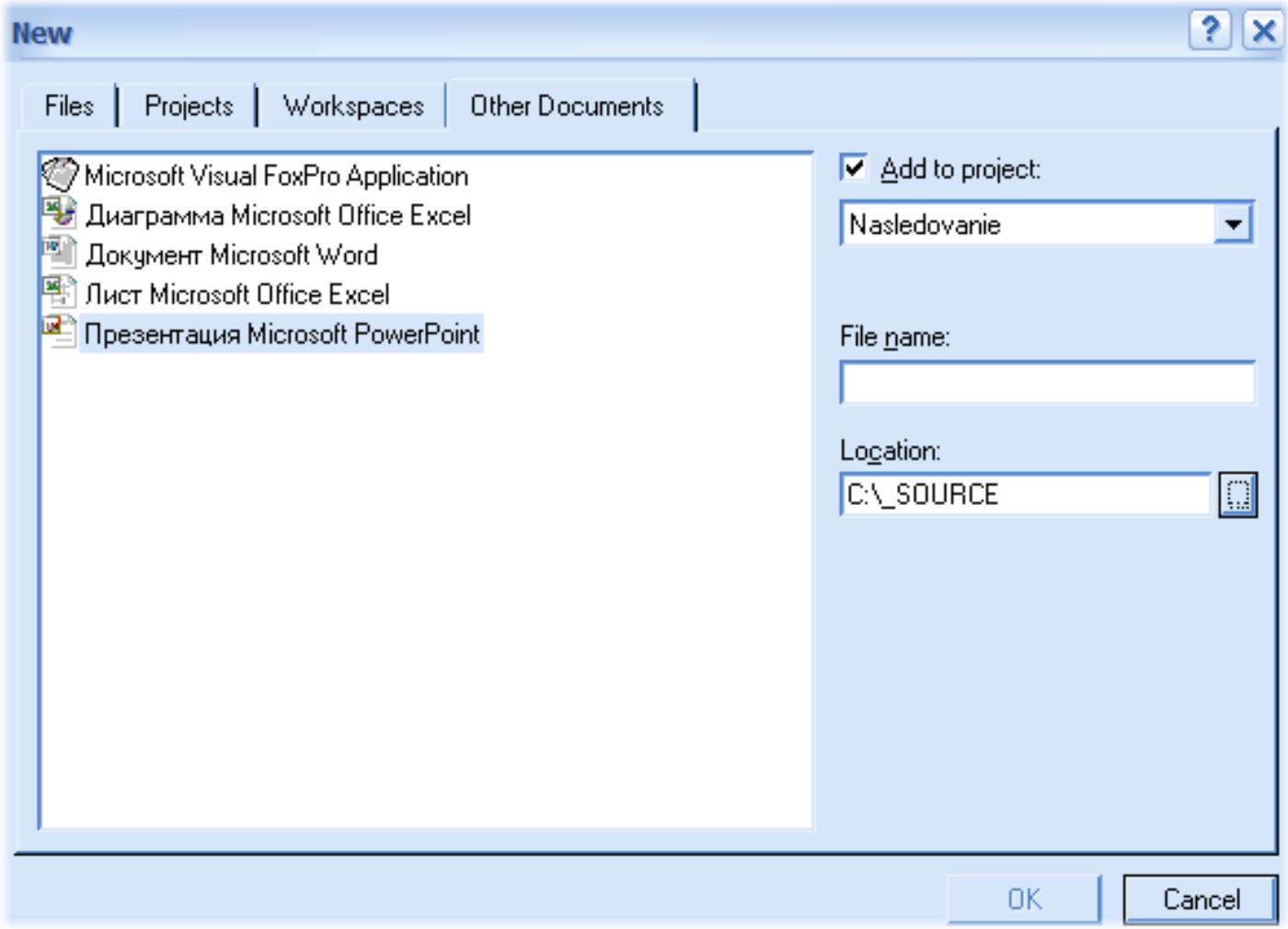
Переключатель *Create New Workspace* – позволяет создать новое рабочее пространство, а переключатель *Add to current workspace* позволяет добавить рабочее пространство к уже существующему, то есть к тому, с которым пользователь работает на данный момент.

Добавление файлов в проект

Для добавления файлов в уже существующий проект существует вкладка *Files*, на которой можно выбрать тип создаваемого файла, его имя и путь к файлу. Опция *Add to project* позволяет добавить файл к текущему активному проекту, т.е. такому, с которым пользователь работает на данный момент.



На вкладке *Other Documents* находится список документов которые также можно добавить в проект, например можно добавить файл документа *Microsoft Word* с текстом лицензионного соглашения.



Среда *Visual C++* позволяет создавать следующие основные типы проектов:

- *MFC AppWizard (exe)* – приложение, поддерживающее полный графический интерфейс пользователя, реализованный на основе классов *MFC*. *Visual C++* автоматически создаст файлы шаблона приложения с соответствующими классами и добавит эти файлы в проект. Выполняемый файл приложения будет иметь расширение *.EXE*

- *MFC AppWizard (dll)* – динамическая *DLL*-библиотека функций, реализованная на основе классов *MFC*. *Visual C++* автоматически создаст файлы шаблона приложения с соответствующими классами и добавит эти файлы в проект. Файл приложения будет иметь расширение *.DLL*

- *Win32 Application* – приложение, поддерживающее полный графический интерфейс пользователя, реализованный или на основе классов *MFC*, или с использованием функций *API*. Выполняемый файл приложения будет иметь расширение *.EXE*

- *Win32 Dynamic-Link Library* – динамическая библиотека функций, реализованная на основе функций *API*. Скомпилированный файл будет иметь расширение *.DLL*

- *Win32 Dynamic-Link Library* – динамическая библиотека функций, реализованная на основе функций *API*. Скомпилированный файл будет иметь расширение *.DLL*

- *MFC ActiveX ControlWizard* – *ActiveX*-элементы управления, реализованные на основе классов *MFC*. *Visual C++* автоматически создаст файлы шаблона приложения с соответствующими классами и добавит эти файлы в проект. Файл приложения будет иметь расширение *.OCX*

- *ISAPI ExtensionWizard* – приложение, поддерживающее работу с *Microsoft Internet Information Server*

- *Database Project* – приложение, поддерживающее средства для доступа к данным из баз данных, реализованное на основе классов *MFC*. *Visual C++* автоматически создаст файлы шаблона приложения с соответствующими классами и добавит эти файлы в проект. Выполняемый файл приложения будет иметь расширение *.EXE*

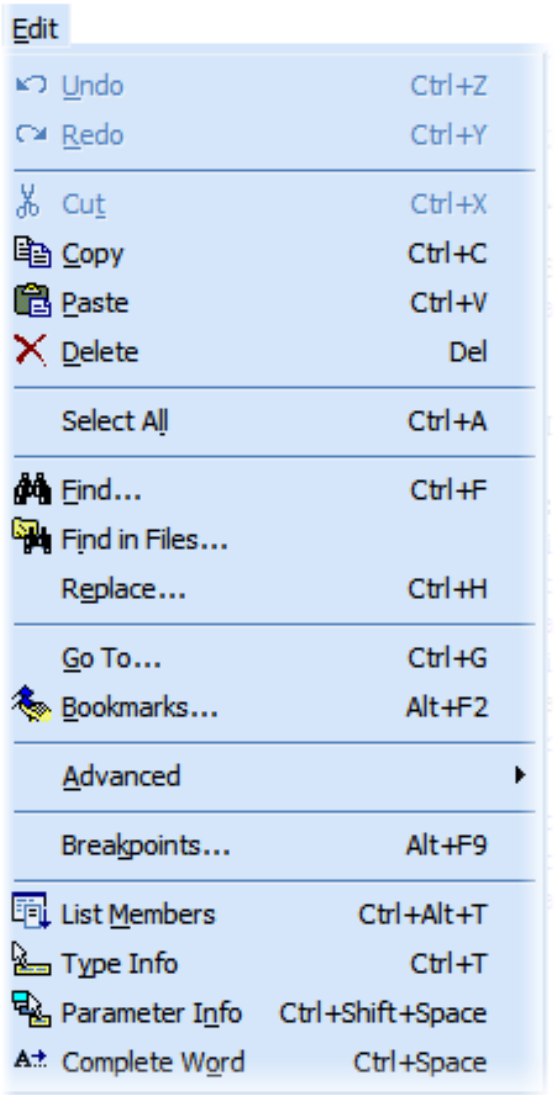
- *Win32 Console Application* – приложение, разработанное с использованием консольных функций *API*, поддерживающих символьный ввод/вывод в окно консоли. Библиотеки времени выполнения *Visual C++* также поддерживают для окна консоли стандартные функции ввода вывода (например, *printf() и scanf()*). Выполняемый файл приложения будет иметь расширение *.EXE*

- *Win32 Static Library* – библиотека функций, включающая в себя все файлы проекта. Скомпилированный файл будет иметь расширение *.LIB*

- *Custom AppWizard* – собственная модификация *MFC AppWizard*, позволяющая разрабатывать другие шаблоны приложений. *Visual C++* автоматически создаст файлы шаблона приложения с соответствующими классами и добавит эти файлы в проект. Скомпилированный файл приложения будет иметь расширение *.AWX*

Меню Edit

Команды этого меню позволяют редактировать текст и проводить поиск по ключевым словам в программном коде, отображаемом в активном окне. Работа этих команд основана на тех же принципах, работа аналогичных команд в большинстве текстов редакторов.

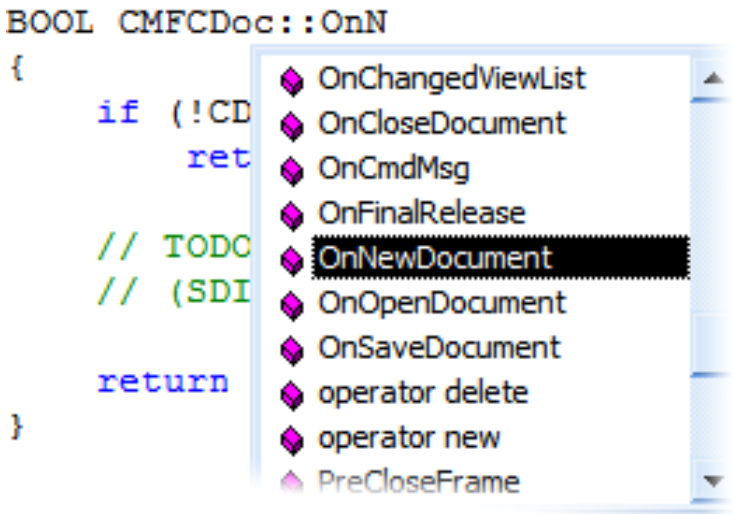


1. *Bookmarks* – команда позволяет помещать закладки в тех местах программы, к которым часто обращает пользователь. После того как закладка будет установлена, можно быстро перейти к ней с помощью команды меню или определенного сочета клавиш

2. *Breakpoints* – установка точки останова (прерывания в любом месте программы

3. *List Members* – список доступных переменных-членов или функций выбранного класса либо структуры

4. *Complete Word* – автоматическое дописывание названия функции или имени переменной, которое пользователь только начал вводить (суфлер кода). Возможно использование сочетания клавиш «Control + Space»

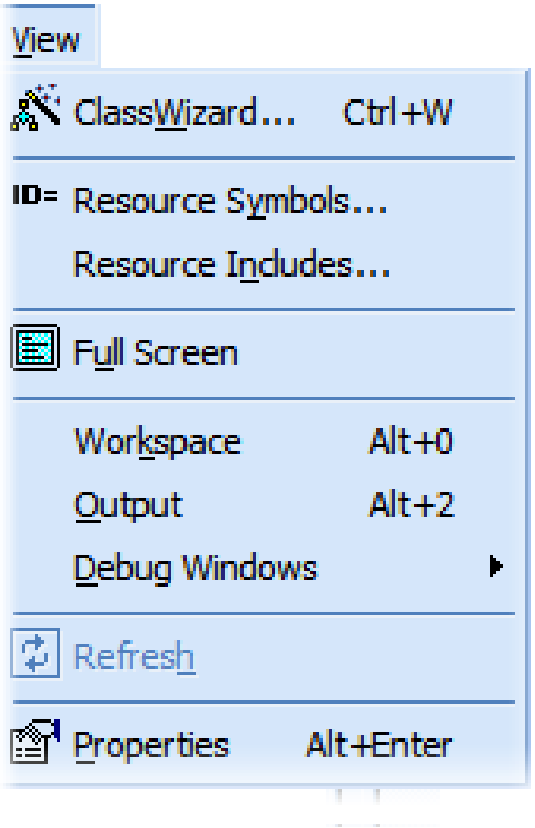


Меню View

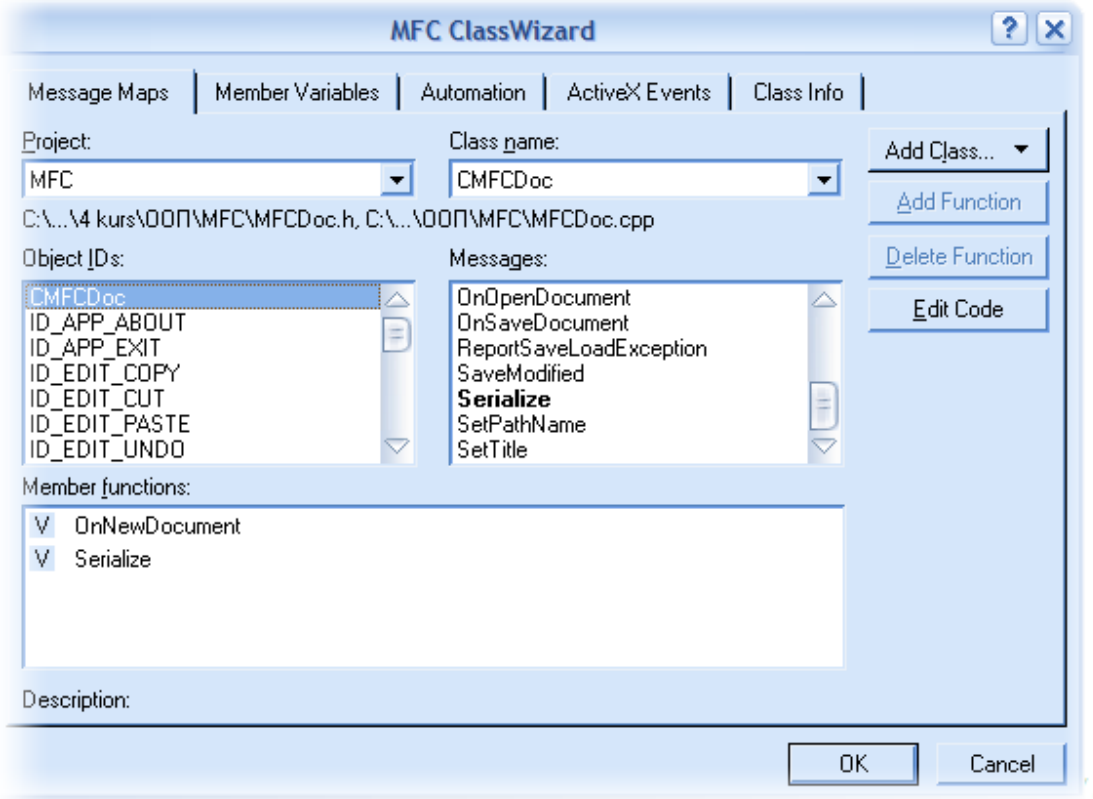
Меню *View* содержит команды, позволяющие настроить внешний вид рабочего пространства.

1. *Class Wizard* – этот мастер облегчит выполнение т

повторяющихся задач, как создание новых классо
и обработчиков сообщений, переопределение
виртуальных функций *MFC* и сбор данных от элем
управления диалоговых окон. Он также может бы
вызван с помощью сочетания клавиш *Control + W*
Class Wizard работает только с приложениями,
использующими библиотеку *MFC*



Окно создания классов для *MFC (Class Wizard):*

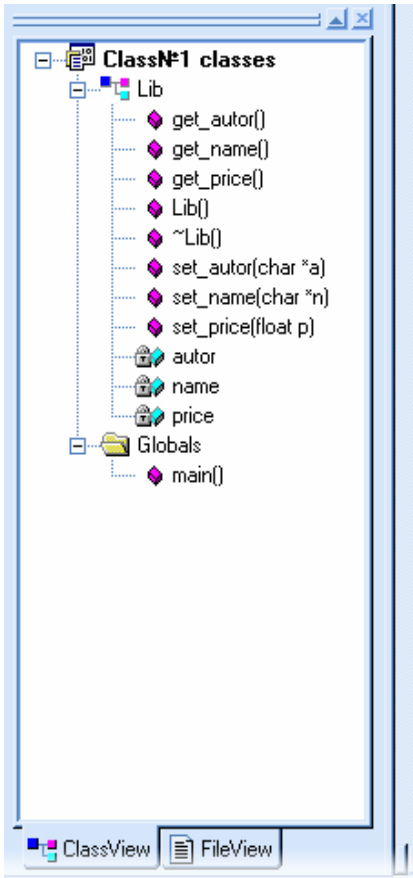


2. *Output* – по команде открывается окно, в котором отображается ход выполнения таких процессов, как компиляция и компоновка программы. В это окно выводятся также все предупреждающие сообщения и сообщения об ошибках, генерируемые компилятором и компоновщиком

Чтобы показать важность окна *Output* разберем простую консольную *Win32* программу.

Необходимо создать один произвольный класс (без наследования) и для него определить атрибуты и методы. После этого скомпилировать программу и проверить ее работоспособность.

В качестве примера разберем создание класса «Библиотека», где в роли атрибутов выступают: название книги, автор и цена, а в качестве методов: модификаторы *set_name, set_autor, set_price* (для установки значений атрибутов экземпляра класса) и селекторы *get_name, get_autor, get_price* (для считывания атрибутов).

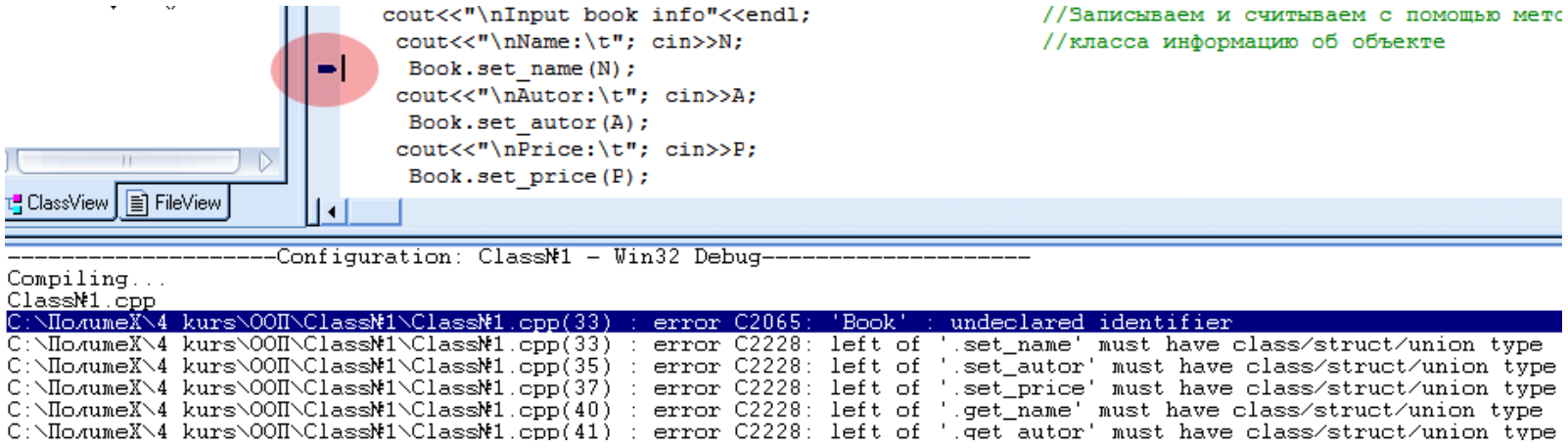



```
public:     //Объявление открытых методов класса
    Lib();
    ~Lib();
        void set_name(char*);       //set - для установки значений экземпллярам класса
        void set_autor(char*);      //методы set_ являются модификаторами
        void set_price(float);
        double get_price();         //get - для считывания значений у экземпляров класса
        char* get_name();           //методы get_ являются селекторами
        char* get_autor();
private:                            //Объявление закрытых элементов класса
    char *name;
    char *autor;
    double price;
};

void main()
{ char *N = new char[128];
     char *A = new char [128];
  float P;



  try
 {
   cout<<"\nInput book info"<<endl;              //Записываем и считываем с помощью методов
    cout<<"\nName:\t"; cin>>N;                   //класса информацию об объекте
     Book.set_name(N);
    cout<<"\nAutor:\t"; cin>>A;
```

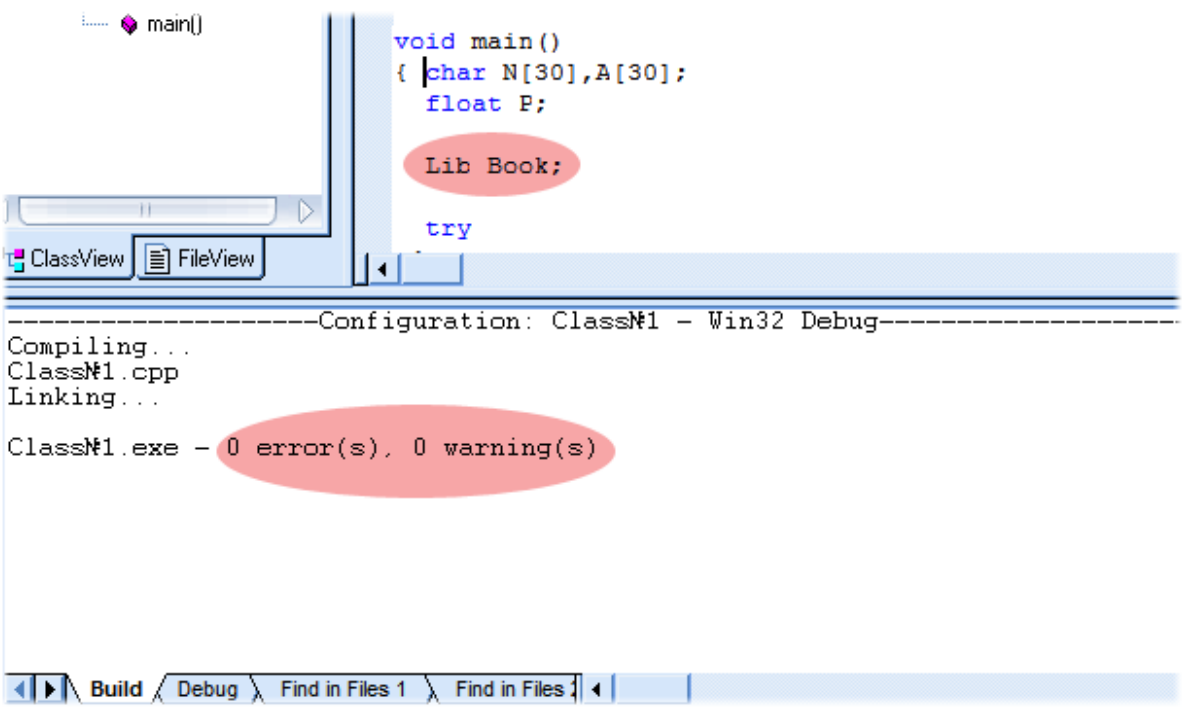
После написания кода программы и при попытке ее компиляции будет выдана ошибка (в окне *Output*), т.к. мы пытаемся работать с классом, не создав его экземпляр:

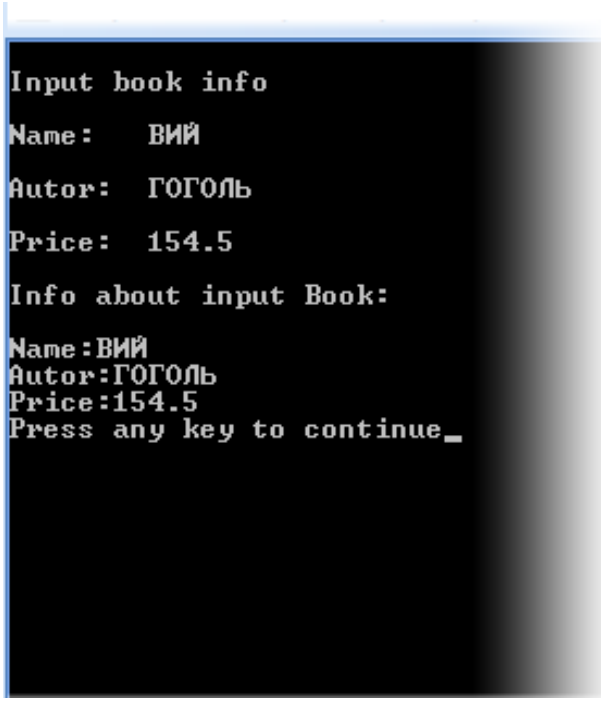
```
--------------------Configuration: Class№1 - Win32 Debug--------------------
Compiling...
Class№1.cpp
C:\ПолинеХ\4 kurs\ООП\Class№1\Class№1.cpp(33) : error C2065: 'Book' : undeclared identifier
C:\ПолинеХ\4 kurs\ООП\Class№1\Class№1.cpp(33) : error C2228: left of '.set_name' must have class/struct/union type
C:\ПолинеХ\4 kurs\ООП\Class№1\Class№1.cpp(35) : error C2228: left of '.set_autor' must have class/struct/union type
C:\ПолинеХ\4 kurs\ООП\Class№1\Class№1.cpp(37) : error C2228: left of '.set_price' must have class/struct/union type
C:\ПолинеХ\4 kurs\ООП\Class№1\Class№1.cpp(40) : error C2228: left of '.get_name' must have class/struct/union type
C:\ПолинеХ\4 kurs\ООП\Class№1\Class№1.cpp(41) : error C2228: left of '.get_autor' must have class/struct/union type
C:\ПолинеХ\4 kurs\ООП\Class№1\Class№1.cpp(42) : error C2228: left of '.get_price' must have class/struct/union type
Error executing cl.exe.

Class№1.exe - 7 error(s), 0 warning(s)
```

При двойном щелчке на строке ошибки (в окне *Output*) курсор в редакторе кода автоматически переместится в то место, где она (ошибка) вызвана:



Если исправить ошибку, создав экземпляр класса прежде чем его использовать, то программа скомпилируется и запустится:

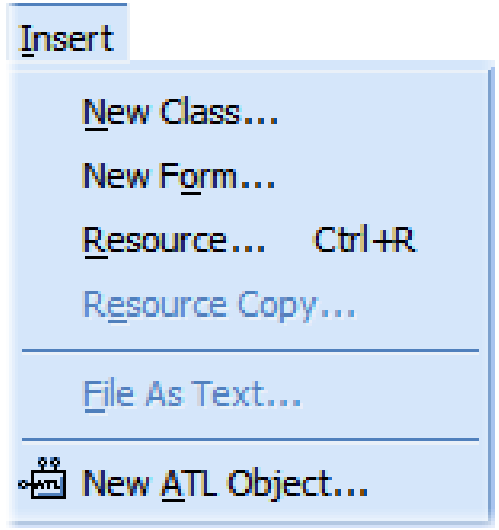




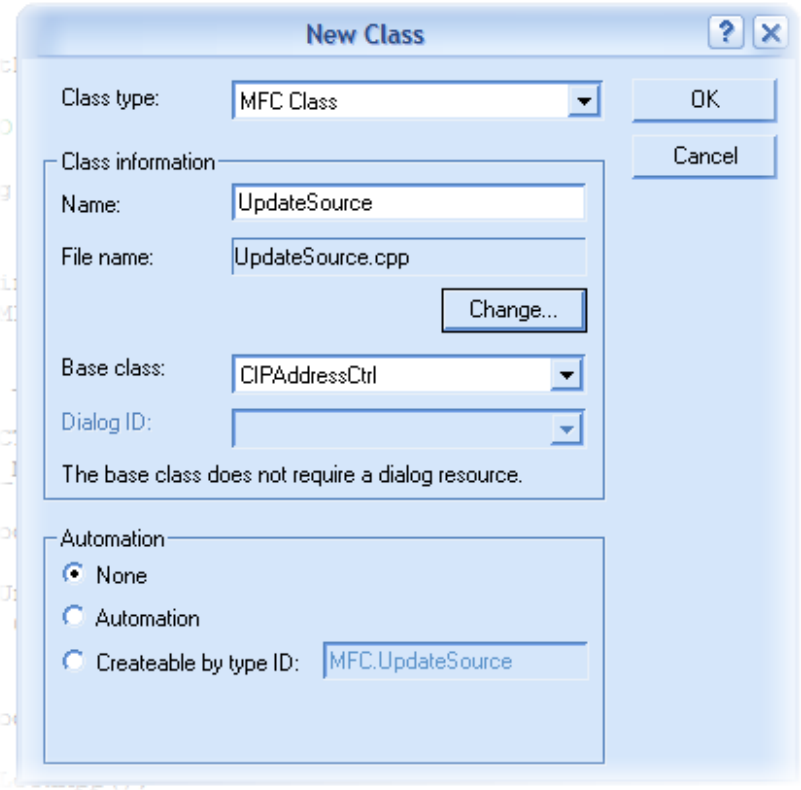
Меню Insert

Меню *Insert* содержит команды, позволяющие вставлять в проект новые файлы, ресурсы, объекты и т.д.

1. *New Class* – при выборе данной команды открывается диалоговое окно *New Class*, в котором можно задать имя нового класса (таковым может быть класс библиотек *MFC* и *ATL* или класс общего назначения) и указать его базовый класс. В результате создается файл заголовков (*.h) и файл реализации (*.cpp) нового класса



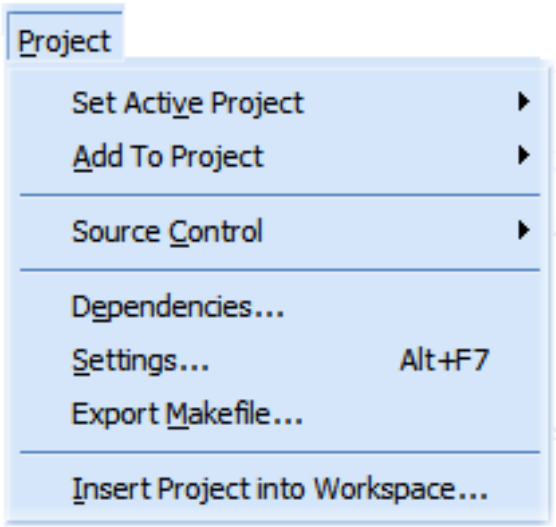
Диалоговое окно создания класса *(New Class)*:



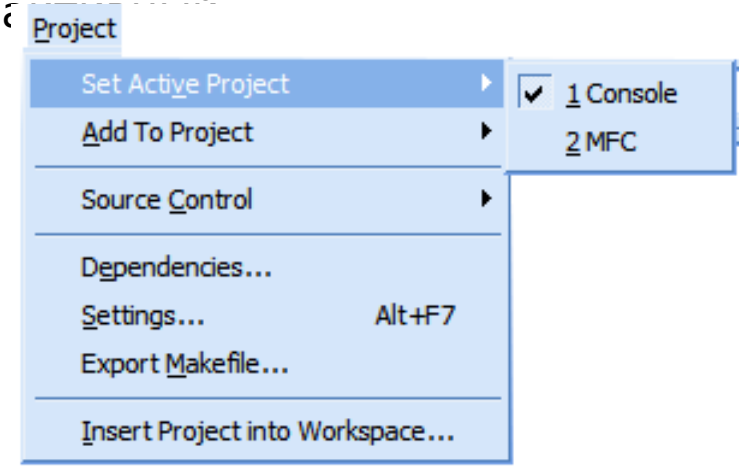
2. *Resource* – эта команда позволяет добавить в проект новые ресурсы, включая горячие клавиши, растровые изображения, указатели мыши, диалоговые окна, значки, HTML-файлы, меню, таблицу строк, панели инструментов и идентификатор версии

Меню Project

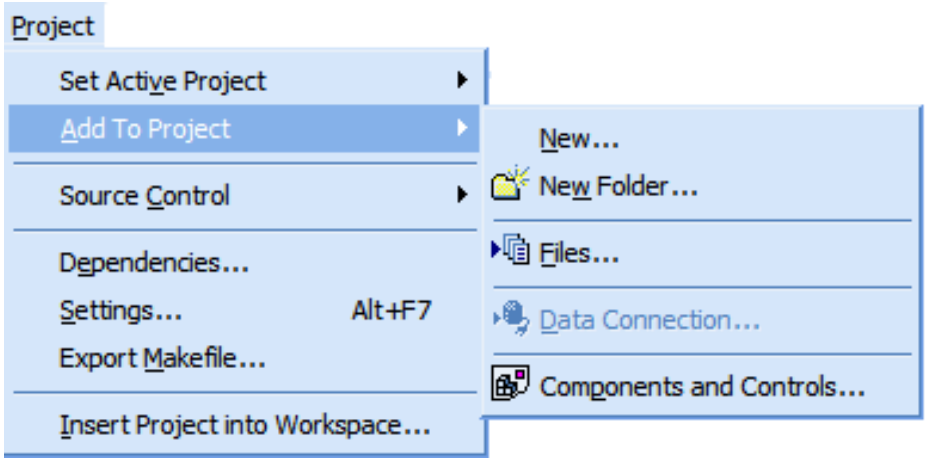
Команды меню *Project* позволяют управлять открытыми проектами



1. *Set Active Project* – в этом подменю отображается список загруженных проектов, из которых можно выбрать а



2. *Add to Project* – это подменю состоит из команд, предназначенных для добавления в проект новых компонентов

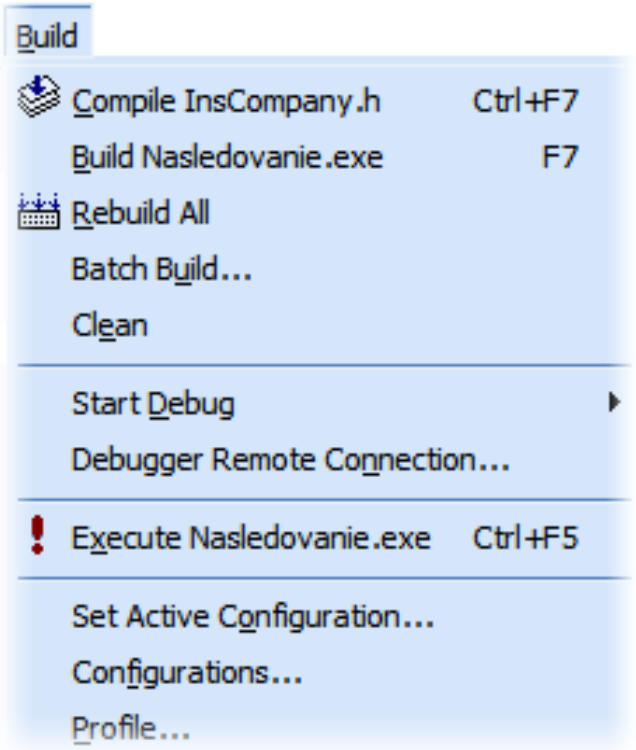


3. *Dependencies* – команда для отображения иерархических связей между множеством подпроектов (в пределах одного проекта)

4. *Export Makefile* – с помощью этой команды можно сохранить в файле всю информацию, необходимую для построения проекта. Файл, созданный с применением этой команды хранит все установки среды *Visual C++.*

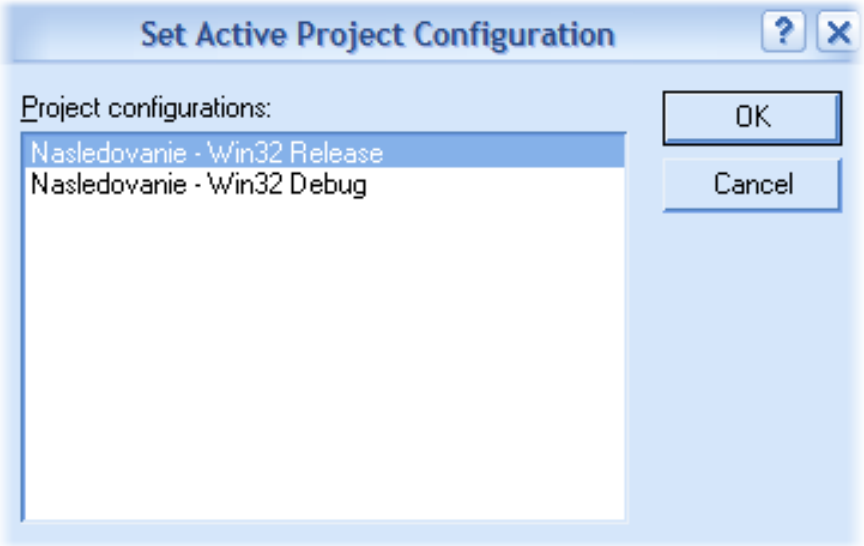
Меню Build

В меню *Build* содержатся всевозможные команды, предназначенные для генерации кода приложения, отладки и запуска созданной программы.



1. *Compile* – выбор этой команды приводит к компиляции содержимого текущего окна
2. *Build* – это команда, которая автоматически проанализирует файлы проекта, компилируя только те из них, которые были созданы позж чем исполняемый файл проекта
3. *Rebuild All* – эта команда компилирует все фа проекта
4. *Batch Build* – эта команда аналогична команд *Build*, но с ее помощью можно обработать сра несколько конфигураций одного проекта

5. *Set Active Configuration* – данная команда позволяет выбрать требуемую конфигурацию проекта.



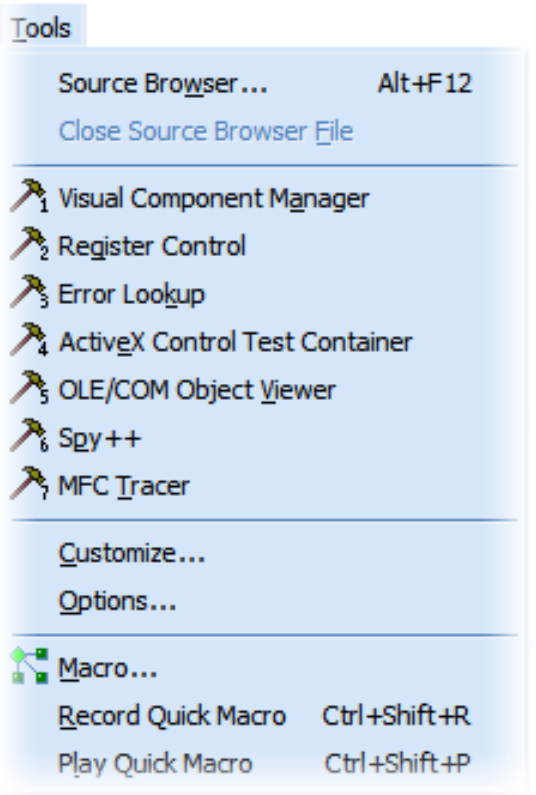
По умолчанию используется конфигурация *Debug* (отладочная версия программы). Если исходный код программы проверен и не содержит ошибок, то пользователь может использовать конфигурацию *Release* (финальная версия программы), при этом из исполняемого файла, в отличие от конфигурации *Debug*, будет удалена отладочная информация и размер **.exe* файла приложения уменьшится в несколько раз.

Соответствующие опции компиляции можно установить и в ручную: *Project -> Settings -> C/C++*

6. *Profile* – данная команда используется для анализа работы программы во время ее выполнения. Чтобы ею можно было воспользоваться, необходимо при создании проекта установить опцию, задающую подключение профилировщика (*Enable profiling* категории *General* вкладки *Link* диалогового окна *Project Settings*). В процессе профилирования в окне *Output* отображается информация, на основании которой вы можете выяснить, какие части программного кода работают эффективно, а какие не выполняются или требуют больших временных затрат.

Меню Tools

В меню *Tools* содержатся команды вызова вспомогательных утилит, программирования макросов и настройки среды *VC+*



1. *Error Lookup* – данная утилита позволяет получити

текст сообщений, связанных с кодами системных
Введите код ошибки в поле *Value*, и в поле *Error M*
автоматически отобразится связанное с ним сообщ

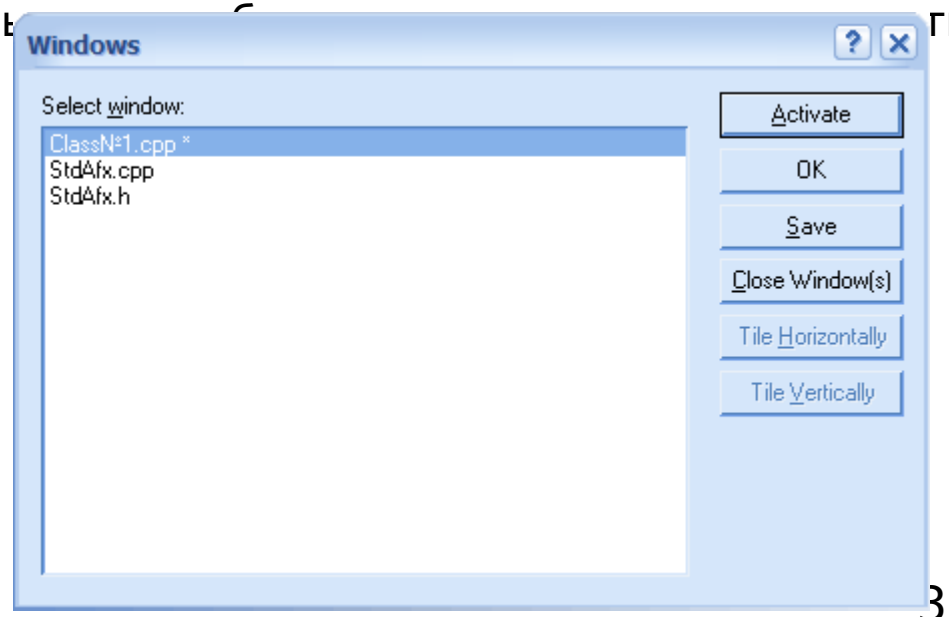
2. *MFC Tracer* – утилита для отладки оконных

приложений, построенных на основе *MFC*. Эта ути
отображает в окне отладки сообщения о
выполнении операций, связанных с использовани
библиотеки MFC, а также предупреждения об
ошибках, если при выполнении приложения проис
какие-либо сбои

3. *Macro... / Record... / Play...* – эти команды используются для создания и воспроизведения макросов на *VB Script*. Макросы представляют собой небольшие процедуры, содержащие команды *VBScript* и не принимающие параметров. Например, можно записать в виде макроса некоторую часто выполняемую последовательность команд , в результате чего для осуществления той же самой задачи достаточно будет ввести простую комбинацию клавиш

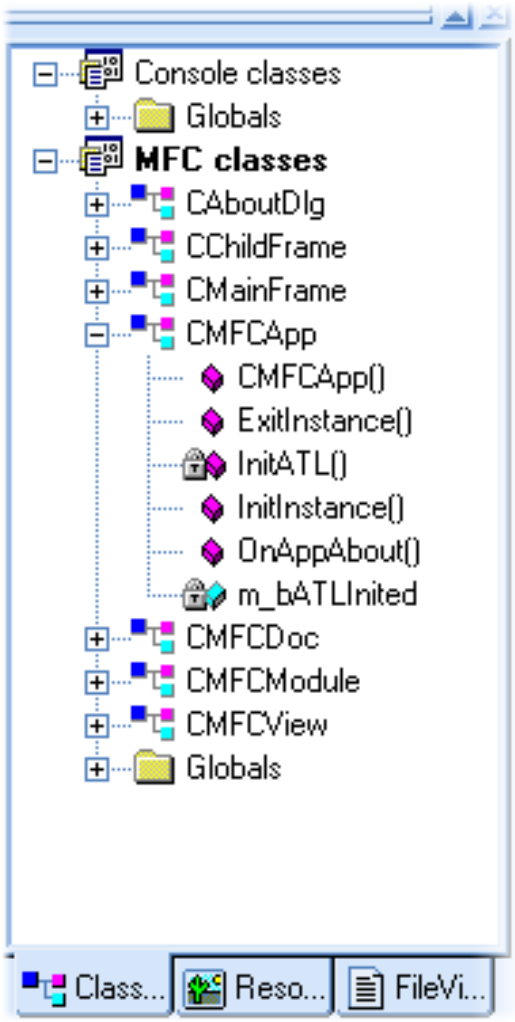
- **Меню Window**

В этом меню содержатся команды для управления положением окон, команды закры гвует обозреватель окон.



Окно проекта (Project Workspace)

В *VC++* существует специальное встроенное окно *Project Workspace*, которое отображает графическое представление рабочего пространства одного или нескольких проектов.

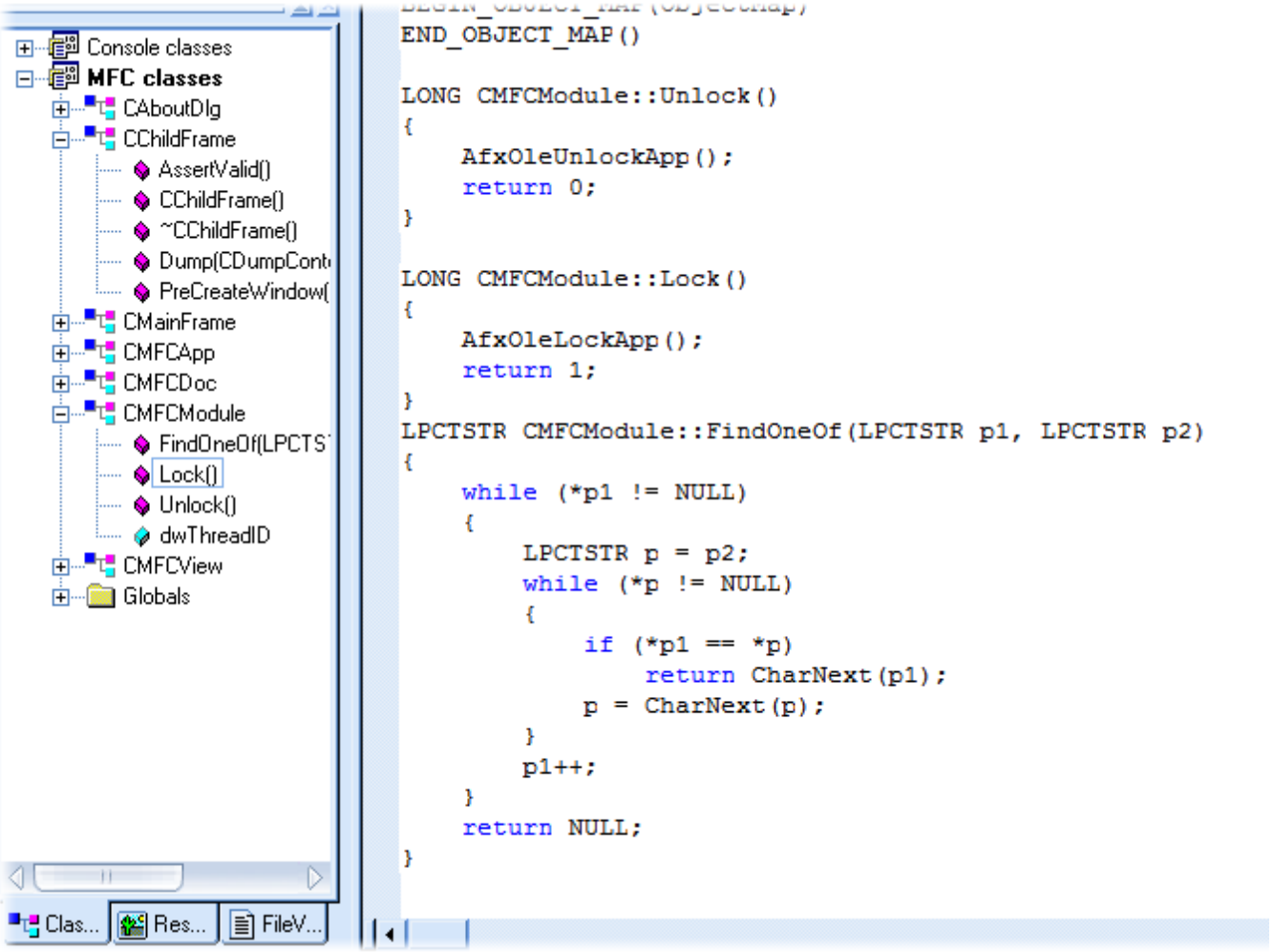


Окно проекта содержит несколько вкладок, позволяющих отображать различные типы информац проекте (или проектах):

- иерархию классов проекта (*Class View*)
- описание ресурсов проекта (*Resource View*)
- входящие в проект файлы (*File View*)
- содержание встроенной справочной системы (*Info V*

Окно кода программы

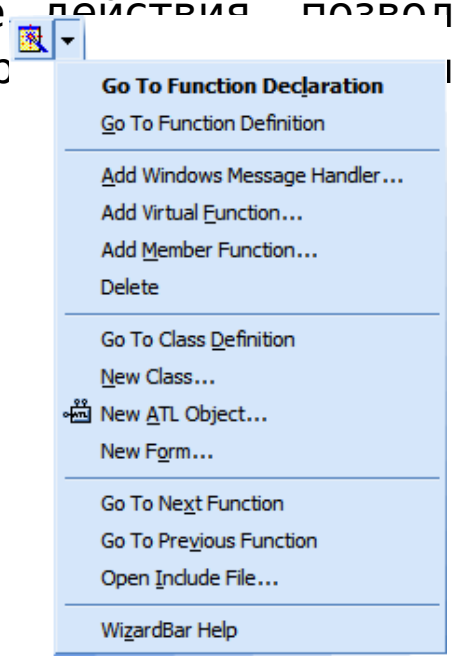
По двойному щелчку на имени класса или атрибута, в окне проекта отображается код заголовочного файла *(*.h)*, а по щелчку на имени метода отображается содержимое файла реализации (**.cpp*)



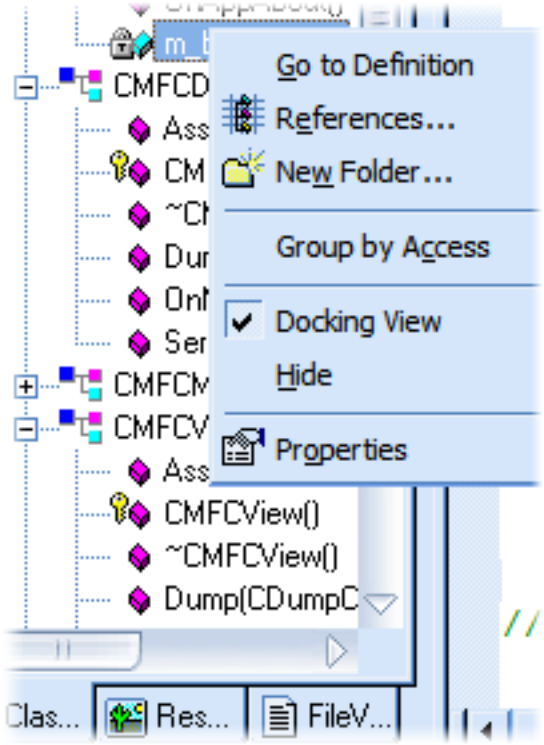
Инструментальная панель Wizard Bar

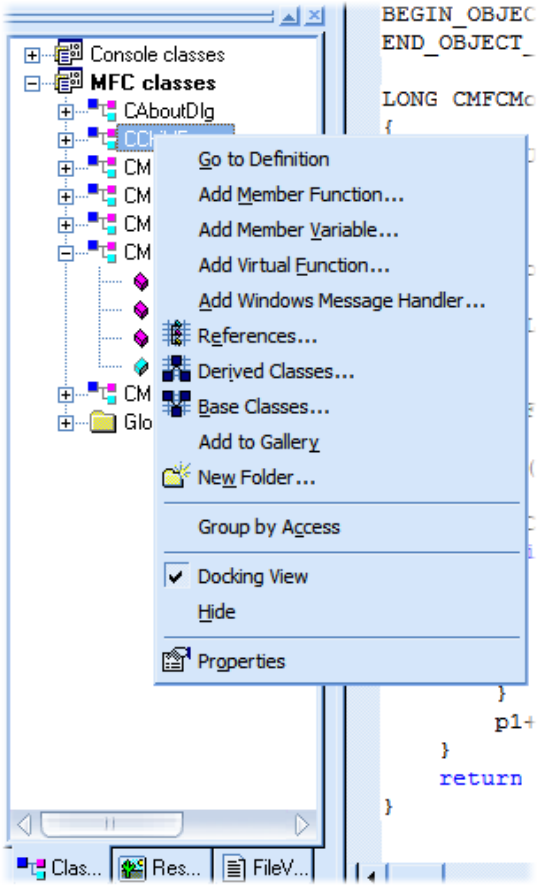
На инструментальной панели *Wizard Bar* отображается информация для активного окна кода программы. С помощью этой панели можно быстро переходить на нужное место в файле, просматривать список доступных сообщений.

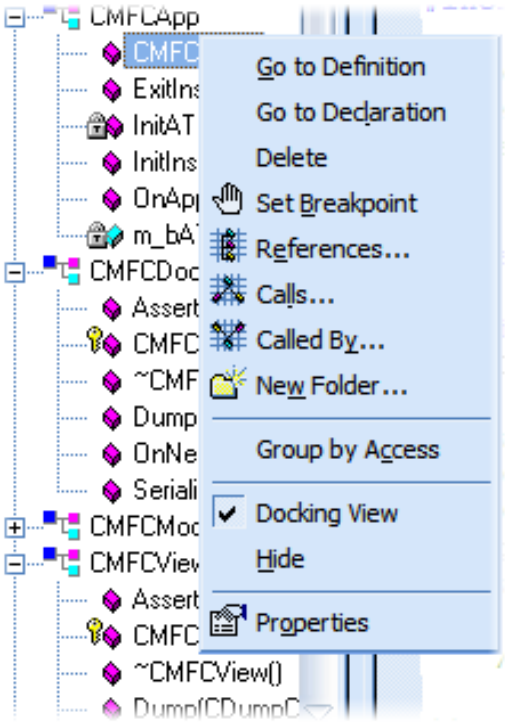
Дополнительно *Wizard Bar* позволяет добавлять новые методы класса (в т.ч. и виртуальные), создавать новые классы, переходить в окно кода для объявления (заголовочный файл) или определения (файл исходного кода) метода, а также выполнять различные другие действия, позволяющие оперативно разрабатывать и редактировать пр проекта приложения.



Функции предоставляемые инструментальной панелью *Wizard Bar*, доступны также в контекстном меню окна проекта, для этого нужно щелкнуть правой кнопкой мыши по имени класса (атрибута, метода).







Вопросы к главе

1. Что в среде *Visual C++* понимается под рабочим пространством, проектом ?
2. Чем отличаются команды *Open* и *Open Workspace* в меню *File* ?
3. Как создать новый проект ?
4. Перечислите основные типы проектов, которые можно создать в *VC++.*
5. Для чего нужна команда *Complete Word* из меню *Edit* ?
6. Перечислите основные функции мастера *Class Wizard*. Со всеми ли типами приложений может работать этот мастер ?
7. Какие функции выполняют команды из меню *Insert* ?
8. Что делает команда *Export Makefile* из меню *Project* ?
9. Чем отличаются команды *Build* и *Rebuild All* из меню *Build* ?
10. Чем отличаются конфигурации *Debug* и *Release* ?
11. Что отображается в окне *Project Workspace* ?
12. Для чего предназначена инструментальная панель *Wizard Bar* ?

II. Основы программирования под Windows

Приложения *Windows* могут создаваться как традиционными методами процедурного программирования на языках C и C++, так и с помощью мощных средств объектно-ориентированного программирования, предоставляемых языком C++.

- **Среда Windows**

  Windows представляет собой графическую многозадачную операционную систему. Программы, разработанные для этой среды, должны соответствовать определенным стандартам и требованиям. Это касается прежде всего внешнего вида окна программы и принципов взаимодействия с пользователями. Чтобы помочь программистам в разработке приложений для *Windows*, были созданы многочисленные системные функции (так называемые API-функции), позволяющие легко добавлять в создаваемые программы контекстные меню, полосы прокрутки, диалоговые окна, значки и многие другие элементы пользовательского интерфейса.

  Windows позволяет работать с различными периферийными устройствами, такими как монитор, клавиатура, мышь, принтер и т.д., вне зависимости от типа самих устройств. Это дает возможность запускать одни и те же приложения на компьютерах с разной аппаратной конфигурацией.

- **Типы данных в Windows**

В *Windows-программах* (и в использующих библиотеку *MFC* в частности) не слишком широко применяются стандартные типы данных из *C* или *C++*, такие как *int* или *char*.* Вместо них используются типы данных, определенные в различных библиотечных файлах.

Наиболее часто используемыми типами являются:

HANDLE – обозначает 32-разрядное целое, используемое в качестве *дескриптора*. Есть несколько похожих типов данных, но все они имеют ту же длину, что и *HANDLE*, и начинаются с литеры H. *Дескриптор* – это просто число, определяющее некоторый ресурс. Например, тип *HWND* обозначает 32-разрядное целое – дескриптор окна. В программах, использующих библиотеку *MFC*, дескрипторы применяются не столь широко, как это имеет место в традиционных программах.

BYTE – обозначает 8-разрядное беззнаковое символьное значение, тип *WORD* – 16-разрядное беззнаковое короткое целое, тип *DWORD* – беззнаковое длинное целое, тип *UINT* - беззнаковое 32-разрядное целое. Тип *LONG* эквивалентен типу *long*. Тип *BOOL* обозначает целое и используется, когда значение может быть либо истинным, либо ложным. Тип *LPSTR* определяет указатель на строку, а *LPCSTR* – константный указатель на строку.

Многозадачная среда

Многозадачность *Windows* состоит в том, что одновременно можно запустить несколько приложений или открыть сразу несколько сеансов работы с одним приложением.

Хотя считается, что приложения выполняются одновременно, в действительности это не так. В текущий момент времени только одно приложение может использовать ресурсы процессора. Многозадачность реализуется за счет того, что процессор переключается между выполняющимися заданиями в течение очень коротких промежутков времени. Приоритеты выполнения одновременно запущенных программ также не одинаковы.

В текущий момент времени активным, то есть принимающим данные от пользователя, может быть только одно приложение, хотя инструкции для процессора могут поступать сразу от всех открытых приложений, независимо от их состояния. Задачу определения приоритетов приложений и распределения времени работы процессора берет на себя *Windows*.

В среде *Windows* все ресурсы динамически распределяются между запущенными приложениями.

Основные понятия многозадачной среды:

1. *Процесс* – это программа которая выполняется; при многозадачности такого типа две или более программы могут выполняться параллельно.

2. *Поток* – это отдельная часть исполняемого кода. В многозадачности данного типа отдельные потоки внутри одного процесса также могут выполняться параллельно.

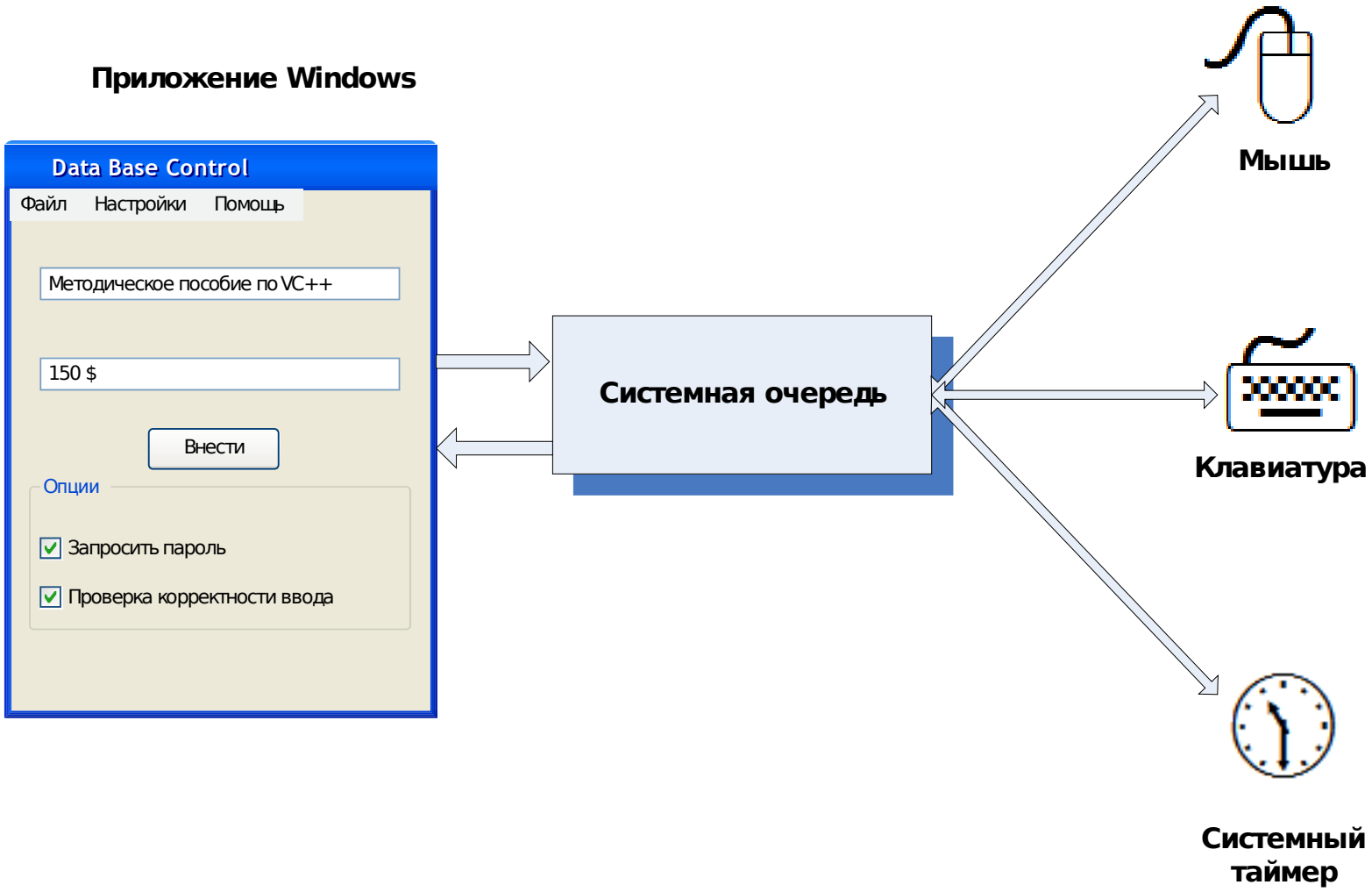
Все процессы имеют, по крайней мере, один поток, но в *Win32-платформах* их может быть несколько. Отсюда можно сделать вывод, что в *Win32-платформах* допускается существование процессов, две или более частей которых выполняются одновременно.

- **Ввод данных посредством очередей**

  В среде *Windows* память компьютера представляет собой совместно используемый ресурс . Таковыми являются и большинство устройств ввода. Поэтому при разработке *Windows-приложений* становятся недоступными функции наподобие *getchar()* языка *C*, считывающие символы непосредственно с клавиатуры, равно как и потоки ввода/вывода языка *C++*.

  В среде *Windows* приложение не может обращаться напрямую к клавиатуре или мыши и получать данные непосредственно от них. Подобная задача выполняется самой *Windows*, которая заносит данные в системную очередь. Из очереди введенные данные распределяются между запущенными программами. Это осуществляется путем копирования сообщений из системной очереди в очереди соответствующих приложений. Затем, как только приложение оказывается готовым принять данные, оно считывает сообщения из своей очереди и распределяет их между открытыми окнами.

Схема работы Windows-приложения с точки зрения сообщений



- **Сообщения**

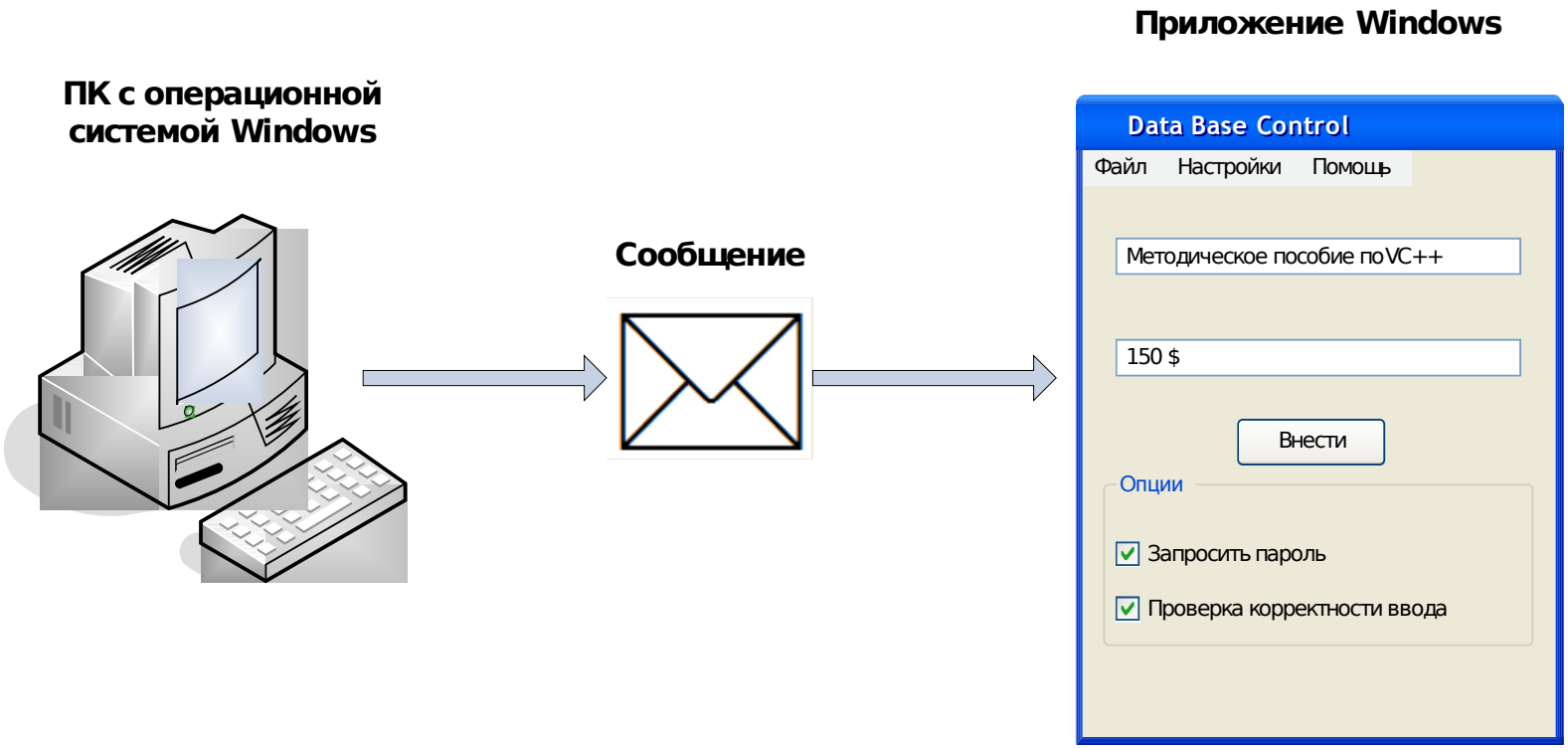
В основе *Windows* лежит механизм генерации и обработки сообщений. С точки зрения приложений, сообщения являются формой уведомления о произошедших событиях, на которые приложение должно (или не должно) каким-то образом реагировать.

Сообщения могут генерироваться автоматически самой *Windows*, например в случае завершения работы системы, когда каждому открытому приложению посылается уведомление о необходимости проверить сохранность данных и закрыть свои окна.

Рассматривая роль сообщений в *Windows*, необходимо отметить, что именно благодаря подсистеме сообщений становится возможной многозадачность *Windows*. Подсистема сообщений позволяет ей распределять время работы процессора между всеми открытыми приложениями. Каждый раз, когда *Windows* посылает сообщение приложению, она на некоторое время открывает этому приложению доступ к процессору. Единственная возможность для приложения получить доступ к процессору — это получить сообщение от *Windows*.

Кроме того, сообщения позволяют приложению прореагировать определенным образом на событие, произошедшее в системе. Это событие могло быть вызвано самим приложением, другим приложением, выполняющимся в это же время в *Windows*, пользователем или операционной системой. Каждый раз, когда происходит то или иное событие, *Windows* оповещает о нем все заинтересованные приложения, открытые в настоящий момент.

Механизм генерации и обработки сообщений



Принципы обработки сообщений

Ни одно приложение в *Windows* не отображает свои данные непосредственно на экране. Точно так же ни одно приложение не обрабатывает напрямую прерывания устройств и не выводит данные непосредственно на печать. Вместо этого приложение вызывает встроенные функции *Windows* и ожидает от системы соответствующих сообщений.

Подсистема сообщений в *Windows* — это средство распределения информации в многозадачной среде. С точки зрения приложения, *сообщение* — это уведомление о некотором произошедшем событии, на которое оно, приложение, должно ответить определенным образом. Такое событие может быть инициировано пользователем, операционной системой или самим приложением.

Особенность этого процесса состоит в том, что приложение должно быть полностью ориентировано на прием и обработку сообщений. Программа должна быть готова в любой момент принять сообщение, определить его тип, выполнить соответствующую обработку и вновь перейти в режим ожидания до поступления следующего сообщения.

Windows открывает приложениям доступ к сотням встроенных функций, которые можно вызывать напрямую или косвенно, посредством библиотек типа *MFC*.

Эти функции содержатся в ряде модулей, таких как *KERNEL*, *GDI* и *USER*:

1. Модуль *KERNEL* отвечает за управление памятью, загрузку и выполнение приложений, а также за распределение системных ресурсов.
2. Модуль *GDI* содержит функции создания и отображения графических объектов.
3. Модуль *USER* отвечает за выполнение всех других функций, обеспечивающих взаимодействие приложений с пользователями и средой *Windows*.

- **Формат сообщений**

Сообщения используются для информирования приложения о том, что в системе произошло то или иное событие. На практике сообщение направляется не столько самому приложению, сколько определенному окну, открытому этим приложением.

Реально в *Windows* существует только один механизм обработки сообщений — системная очередь сообщений. Но каждое выполняющееся приложение организовывает и свою очередь.

Функции модуля *USER*, в частности, ответственны за передачу сообщений из системной очереди в очередь конкретного приложения. В последней накапливаются сообщения, адресованные любому окну, открытому данным приложением.

Независимо от типа все сообщения характеризуются четырьмя параметрами: *дескриптором окна*, которому адресуется данное сообщение, *типом сообщения* и еще двумя 32-разрядными параметрами.

Дескрипторы широко используются в приложениях *Windows*. *Дескриптором* называется уникальный номер, который присваивается всем системным объектам, таким как окна, элементы управления, меню, значки, а также областям памяти, устройствам вывода и т.д.

- **Управление памятью**

Одним из наиболее важных ресурсов системы является память компьютера.

Если в системе одновременно запущено несколько приложений, то они должны иметь возможность координировать свою работу таким образом, чтобы не вызывать конфликтов при распределении системных ресурсов. Так, в результате многократного открытия и закрытия программ память компьютера может оказаться фрагментированной. *Windows* обладает встроенной подсистемой дефрагментации памяти, организующей перемещение и объединение блоков памяти. Если же размер кода запускаемого приложения превышает объем свободной памяти, появляется опасность исчерпания памяти компьютера. *Windows* способна автоматически выгружать из памяти неиспользуемые в данный момент фрагменты приложений и загружать их повторно с диска при обращении к ним.

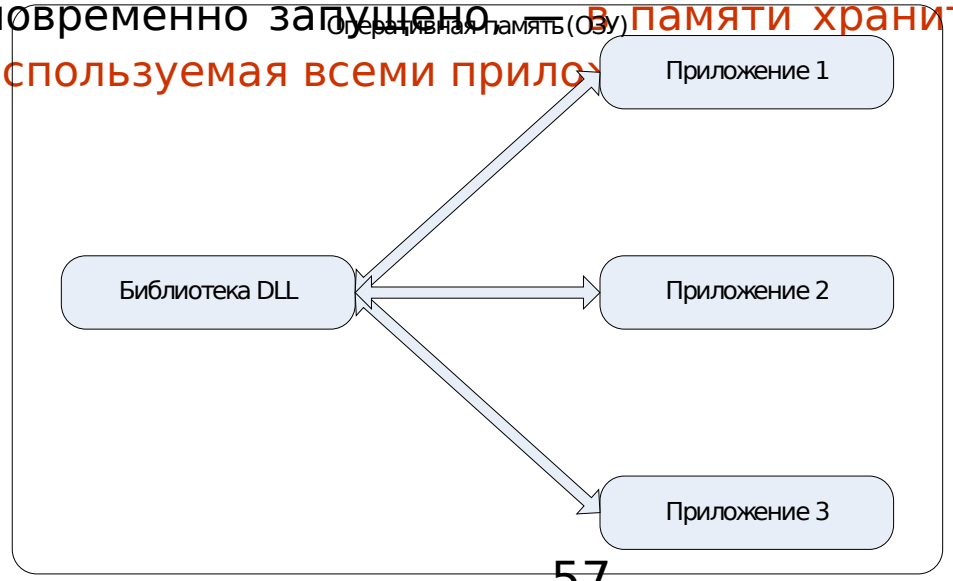
Приложения *Windows* могут совместно использовать функции, хранящиеся в отдельных исполняемых файлах общего доступа. Такие файлы называются *библиотеками динамической компоновки* (*DLL — dynamic link libraries*). *Windows* содержит встроенный механизм компоновки таких библиотек с программами на этапе выполнения.

Библиотеки динамической компоновки

Функциональные возможности *Windows* в большой мере обеспечиваются за счет использования библиотек динамической компоновки *(DLL)*. В частности, благодаря этим библиотекам к ядру операционной системы добавляется графический пользовательский интерфейс.

DLL-файлы содержат функции, которые подключаются к программе во время ее выполнения (динамически), а не во время компиляции (статически).

Все библиотеки *Windows* компонуются динамически. То есть функции из этих библиотек не копируются в исполняемые файлы, а вызываются во время выполнения программы, что позволяет экономить ресурсы памяти. Не важно, сколько приложений одновременно запущено — в памяти хранится только одна копия библиотеки, используемая всеми прило



- **Вызов системных функций**

  Как известно, *Windows* содержит сотни встроенных функций, таких как *DispatchMessage(), PostMessage(), RegisterWindowMessage(), SetActiveWindow()* и т.д. При создании приложений на *C++* с использованием библиотеки *MFC* большинство этих функций инкапсулируются в базовых классах и выполняются автоматически.

- **Функция WinMain**

  Все *Windows-программы* начинают свое выполнение с вызова функции *WinMain.*

  Эта функция отвечает за следующее:

1. создание и инициализацию цикла обработки сообщений (который имеет доступ к очереди сообщений приложения);
2. начальную инициализацию приложения;
3. регистрацию класса окна приложения;
4. завершение выполнения программы.

При традиционном методе программирования вызов этой функции нужно делать явно. С использованием библиотеки *MFC* такая необходимость отпадает, т.к. в *MFC* есть класс *CWinApp*, который и включает главную функцию программы. Естественно, что в приложении она может быть только одна.

Пример:

```
  #include <afxwin.h>          // MFC; Основные и стандартные компоненты
  …
  class CMyApp : public CWinApp // создали класс CMyApp, производный
                                //  от  CWinApp  унаследовав  все  его
                               // свойства и методы (в т.ч. WinMain())
{
  public:
   CMyApp();                              // конструктор по умолчанию
      virtual BOOL InitInstance();   // стандартная инициализация
};
  …
```

Этапы создания приложения под Windows

Разработку приложений *Windows* можно разделить на несколько общих этапов, таких как:

1. Создание функции *WinMain* и других базовых функций. При использовании *MFC* эти действия выполняются автоматически в классе *CWinApp*.
2. Создание меню и других необходимых ресурсов, включение их в файл сценария ресурсов.
3. Создание с помощью редактора ресурсов уникальных указателей мыши, значков и других растровых изображений (Не обязательный).
4. Создание с помощью редактора ресурсов дополнительных диалоговых окон (Не обязательный).
5. Компиляция проекта.

Венгерская нотация

Некоторые переменные при написании программ под *ОС Windows* обозначаются своеобразными именами (например имя *szCmdLine*).

Многие программисты при написании приложений под *ОС Windows* используют соглашения по именованию переменных, названные условно Венгерской нотацией. Ее суть заключается в следующем: имя переменной начинается со строчных буквы или букв, которые характеризуют тип данных переменной. Например, префикс *sz* в *szCmdLine* означает, что строка завершается нулем (*string terminated by zero*). Префикс *h* в *hInstance* и *hPrevInstance* означает *handle*; префикс *i* в *iCmdShow* означает целое (*integer*).

При обозначении переменных структуры (класса) удобно пользоваться именем самой структуры (или аббревиатурой имени структуры) и строчными буквами, используя их либо в качестве префикса имени переменной, либо как имя переменной в целом. Например, переменная *msg* относится к структуре (классу) типа MSG; *wndclass* — к структуре типа *WNDCLASSEX*.

Венгерская нотация помогает избегать ошибок в программе еще до ее компоновки. Поскольку имя переменной описывает и саму переменную и тип ее данных, то намного снижается вероятность введения в программу ошибок, связанных с несовпадением типа данных у переменных.

В следующей таблице представлены основные префиксы переменных:

Префикс	Тип данных
c	Символ
by	Byte (беззнаковый символ)
n	Короткое целое
i	Целое
x, y	Целое (используется в качестве координат x и y)
b	BOOL (булевое целое)
w	WORD (беззнаковое короткое целое)
l	LONG (длинное целое)
dw	DWORD (беззнаковое длинное целое)
fn	Функция
s	Строка
sz	Строка, завершаемая нулем (‘\0’)
h	Дескриптор объекта

Вопросы к главе

1. Расскажите, в общих чертах, что представляет собой *ОС Windows* ?
2. Перечислите основные типы данных, используемые в *ОС Windows* ?
3. Что такое *дескриптор* и для чего он используется ?
4. Что понимается под *многозадачностью* ?
5. Дайте определение терминам *процесс* и *поток* ?
6. Что такое *сообщение* и для чего оно предназначено ? Кем, или чем, могут генерироваться сообщения в *ОС Windows* ?
7. Какие функции выполняют модули *KERNEL*, *GDI*, *USER* ?
8. Что такое *библиотека динамической компоновки* ? Какие функции она выполняет ?
9. Для чего предназначена и за что отвечает функция *WinMain* ?
10. В чем суть Венгерской нотации ?

III. Библиотека MFC
Создание приложений под ОС Windows

- **Основные сведения о библиотеке MFC**

Приложения *Windows* просты в использовании, но создавать их довольно сложно.

Чтобы облегчить труд программистов, специалисты фирмы *Microsoft* разработали библиотеку *MFC (Microsoft Foundation Classes).*

Библиотека *MFC* существенно облегчает программирование в среде *Windows*. Используя готовые классы, можно гораздо быстрее и проще решать многие задачи. То есть можно дорабатывать классы или создавать новые, производные от существующих.

Классы библиотеки *MFC* используются как для управления объектами *Windows*, так и для решения определенных общесистемных задач. Например, в библиотеке имеются классы для управления файлами, строками, временем, обработкой исключений и другие.

По сути, в *MFC* представлены практически все функции Windows *API*. В библиотеке имеются средства обработки сообщений, диагностики ошибок и другие средства, обычные для приложений *Windows*.

Основные преимущества использования библиотеки MFC

- Представленный набор функций и классов отличается логичностью и полнотой. Библиотека *MFC* открывает доступ ко всем часто используемым функциям Windows *API*, включая функции управления окнами приложений, сообщениями, элементами управления, меню, диалоговыми окнами, объектами *GDI* (*Graphics Device Interface* — интерфейс графических устройств), такими как шрифты, кисти, перья и растровые изображения, функции работы с документами и многое другое

- Функции *MFC* легко изучать. Специалисты *Microsoft* приложили все усилия для того, чтобы имена функций *MFC* и связанных с ними параметров были максимально близки к их эквивалентам из Windows *API*. Благодаря этому программисты легко смогут разобраться в их назначении

- Программный код библиотеки достаточно эффективен. Скорость выполнения приложений, основанных на *MFC*, будет примерно такой же, как и скорость выполнения приложений, написанных на *C* с использованием стандартных функций Windows *API*, а дополнительные затраты оперативной памяти будут весьма незначительными

- *MFC* содержит средства автоматического управления сообщениями. Библиотека *MFC* устраняет необходимость в организации цикла обработки сообщений — распространенного источника ошибок в *Windows-приложениях*. В *MFC* предусмотрен автоматический контроль за появлением каждого сообщения. Вместо использования стандартного блока *switch/case* все сообщения *Windows* связываются с функциями-членами, выполняющими соответствующую обработку

- *MFC* позволяет организовать автоматический контроль за выполнением функций. Эта возможность реализуется за счет того, что можно записывать в отдельный файл информацию о различных объектах и контролировать значения переменных-членов объекта в удобном для понимания формате.

- *MFC* имеет четкий механизм обработки исключительных ситуаций. Библиотека *MFC* была разработана таким образом, чтобы держать под контролем появление таких ситуаций. Это позволяет объектам *MFC* восстанавливать работу после появления ошибок типа «Out of memory» (нехватка памяти), неправильного выбора команд меню или проблем с загрузкой файлов либо ресурсов

- *MFC* может использоваться совместно с подпрограммами, написанными на языке *C*. Важной особенностью библиотеки *MFC* является то, что она может "сосуществовать" с приложениями, основанными на *Windows API*. В одной и той же программе программист может использовать классы *MFC* и вызывать функции Windows *API*. Такая прозрачность среды достигается за счет согласованности программных обозначений в обеих архитектурах. Другими словами, файлы заголовков, типы и глобальные константы *MFC* не конфликтуют с именами из *Windows API*. Еще одним ключевым моментом, обеспечивающим такое взаимодействие, является согласованность механизмов управления памятью

- *MFC* может быть использована для создания программ, работающих в среде *MS-DOS*. Библиотека *MFC* была создана специально для разработки приложений в среде Windows. В то же время многие классы предоставляют объекты, часто используемые для ввода/вывода файлов и манипулирования строковыми данными. Такие классы общего назначения могут применяться в приложениях как *Windows*, так и *MS-DOS*

- *MFC* обеспечивает динамическое определение типов объектов. Это чрезвычайно мощное программное средство, позволяющее отложить проверку типа динамически созданного объекта до момента выполнения программы. Благодаря этому можно свободно манипулировать объектами, не заботясь о предварительном описании типа данных. Поскольку информация о типе объекта возвращается во время выполнения программы, программист освобождается от целого этапа работы, связанного с типизацией объектов.

Минимальная программа для *Windows* состоит из двух функций: функции WinMain и функции окна.

На некотором псевдоязыке программу для *Windows* можно записать следующим образом:

```
WinMain(список аргументов)
   {    Создание класса окна
        Создание экземпляра класса окна
        Пока не произошло необходимое для выхода событие
        {
           Выбрать из очереди сообщений очередное сообщение
            Передать сообщение оконной функции
             Возврат из программы
        }
   }
WindowsFunction(список аргументов)
   {
   Обработать полученное сообщение
   Возврат
   }
```

- **Основные сведения об Интерфейсе прикладного программирования (API)**

API (Application Programming Interface) — это набор функций, предоставляющих программисту возможность создавать программы (приложения) для 32-разрядных операционных систем семейства *Windows (Win 95/98, Win NT/2k/XP)*. Естественно, что эти платформы разнятся между собой, но набор функций, составляющих *API*, для них один и тот же.

Таким образом, *Windows API* (или сокращенно *WinAPI*) есть не что иное, как просто набор функций, формирующих интерфейс между приложением и компьютером в целом.

Рассмотрим пример программы, написанной с использованием *WinAPI*-функций и реализованной в среде *MS Visual C++*, создающей на экране окно и реагирующей на некоторые события.

Ниже приведен полный исходный код программы.

```
#include "stdafx.h"
#include "windows.h"
#include "windowsx.h"

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM,LPARAM);
int APIENTRY WinMain(
HINSTANCE hInstance,
HINSTANCE hPrevInstance,
LPSTR lpCmdLine,
int nCmdShow )
{
HWND hwnd;
MSG msg;
WNDCLASS w;
memset(&w,0,sizeof(WNDCLASS));
w.style = CS_HREDRAW | CS_VREDRAW;
w.lpfnWndProc = WndProc;
w.hInstance = hInstance;
```

```
        w.hbrBackground = GetStockBrush(WHITE_BRUSH);
        w.lpszClassName = "API WINDOW";
        RegisterClass(&w);
        hwnd = CreateWindow("API WINDOW", "TEST API",
    WS_OVERLAPPEDWINDOW,
        10,10,600,480,NULL,NULL,hInstance,NULL);
        ShowWindow(hwnd,nCmdShow);
        UpdateWindow(hwnd);

      while(GetMessage(&msg,NULL,0,0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
      return msg.wParam;
}
```

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT Message, WPARAM  wparam,LPARAM
lparam)
{
   if ( Message == WM_LBUTTONUP )
   {    Beep(39,100);
   }
   if ( Message == WM_DESTROY )
   {    PostQuitMessage(0);
   return 0;
   }
   if ( Message == WM_CLOSE )
   {    MessageBox(NULL, "Завершение программы","Инфо",MB_OK |
   MB_ICONINFORMATION);
   }
   if ( Message == WM_CREATE )
   {    MessageBox(NULL, "Открытие программы","Инфо",MB_OK |
   MB_ICONINFORMATION);
   }    return DefWindowProc(hwnd, Message, wparam, lparam);
}
```

Разберем данный код.

Написание программы осуществляется с подключения необходимых заголовочных файлов, содержащих прототипы функций, описание структур, классов, констант и иные сведения:

```
#include "stdafx.h"
#include "windows.h"
#include "windowsx.h"
```

Затем описываем прототип функции *WndProc* («оконная процедура», функция обработки сообщений для объектов зарегистрированного оконного класса, иными словами функция посылающая сообщения окну):

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT Message, WPARAM wparam,LPARAM lparam);
```

Рассмотрим параметры функции.

HWND ***hwnd*** — *дескриптор* объекта окна, для которого предназначено сообщение, если создано несколько объектов данного оконного класса, то по этому *дескриптору* выбирается нужный объект; для главного окна приложения, которое существует в одном экземпляре его можно проигнорировать.

UINT ***Message*** — код сообщения (беззнаковое целое).

WPARAM ***wParam*** и **LPARAM** ***lParam*** — 32-х битные целые, в которые передаются параметры сообщения.

В исходном коде программы отсутствуют инструкции для непосредственного вызова функции **WndProc** — **WndProc** вызывается только из *Windows*. Однако, в **WinMain** имеется ссылка на **WndProc** (**w.lpfnWndProc = WndProc**), поэтому эта функция описывается в самом начале программы, еще до определения **WinMain**.

Далее идет описание функции *WinMain* — точки входа в программу для *Windows*. Все *Windows-программы* начинают свое выполнение с вызова функции *WinMain.* Эта функция использует последовательность вызовов *WINAPI* и, по своему завершению, возвращает операционной системе *Windows* целое.

```
int  APIENTRY WinMain  (
     HINSTANCE hInstance,
     HINSTANCE hPrevInstance,
     LPSTR lpCmdLine,
     int nCmdShow
    )
```

Обратите внимание на ключевое слово *APIENTRY* перед телом функции *WinMain*. Это слово указывает компилятору на необходимость сгенерировать перед выполнением этой функции специальный пролог и эпилог необходимый для функции, в которой запускается и завершается программа. Если это ключевое слово будет отсутствовать, то программа будет сгенерированна неправильно.

В данной функции присутствует четыре параметра, рассмотрим их подробнее:

`HINSTANCE hInstance` — описатель экземпляра (*instance handle*). Это уникальное число, идентифицирующее программу, когда она работает под *Windows*. Может так случиться, что пользователь запустит под *Windows* несколько копий одной и той же программы. Каждая копия называется "экземпляром" и у каждой свое значение **`hInstance`**. Описатель экземпляра можно сравнить с "идентификатором задачи" или "идентификатором процесса".

`HINSTANCE hPrevInstance` — предыдущий экземпляр (*previous instance*). Этот параметр перешел в *Win32* в целях совместимости с предыдущими версиями *Win16*. Он не используется и всегда равен NULL.

LPSTR lpCmdLine — это указатель на оканчивающуюся нулем строку, в которой содержатся любые параметры, переданные в программу из командной строки. Если таких параметров нет то значение равно NULL.

int nCmdShow — число, показывающее, каким должно быть выведено на экран окно в начальный момент. В большинстве случаев это число равно 1 или 7. Эти числа соответствуют идентификаторам *SW_SHOWNORMAL* (нормальный размер окна) и *SW_SHOWMINNOACTIVE* (окно свернуто).

Затем начинаем описывать тело функции *WinMain.*

```
HWND hwnd;
MSG msg;
WNDCLASS w;
```

Идентификатор (*дескриптор*) **HWND hwnd** нужен для описания окна, а также для ссылки на него из других функций *Windows.*

MSG msg — описывает структуру сообщения **msg**. В *ОС Windows* данная структура имеет вид:

```
typedef struct {
      HWND hwnd;
      UINT message;
      WPARAM wParam;
      LPARAM lParam;
      DWORD time;
      POINT pt; }  MSG, *PMSG;
```

WNDCLASS w — описывает структуру, содержащую необходимые данные о классе окна. В *ОС Windows* данная структура имеет вид:

```
typedef struct {
     UINT style;
     WNDPROC lpfnWndProc;
     int cbClsExtra;
     int cbWndExtra;
     HINSTANCE hInstance;
     HICON hIcon;
     HCURSOR hCursor;
     HBRUSH hbrBackground;
     LPCTSTR lpszMenuName;
     LPCTSTR lpszClassName;
}    WNDCLASS, *PWNDCLASS;
```

Далее устанавливаем параметры для экземпляра класса окна:

```
w.style = CS_HREDRAW | CS_VREDRAW;
w.lpfnWndProc = WndProc;
w.hInstance = hInstance;
w.hbrBackground = GetStockBrush(WHITE_BRUSH);
w.lpszClassName = "API WINDOW";
w.lpszMenuName = NULL;
```

Инструкция `w.style = CS_HREDRAW | CS_VREDRAW` показывает, что все окна, созданные на основе данного класса должны целиком перерисовываться при изменении горизонтального (`CS_HREDRAW`) или вертикального (`CS_VREDRAW`) размеров окна.

Инструкция `w.lpfnWndProc = WndProc` устанавливает `WndProc` как оконную процедуру данного окна. Эта процедура будет обрабатывать все сообщения для всех окон, созданных на основе данного класса окна (приставка `lpfn` означает «длинный указатель на функцию»).

Инструкция `w.hInstance = hInstance` устанавливает идентификатор экземпляра программы, регистрирующей класс.

Инструкция `w.hbrBackground = GetStockBrush(WHITE_BRUSH)` устанавливает фон рабочей области окна в белый цвет.

Инструкция `w.lpszClassName = "API WINDOW"` устанавливает имя класса окна.

Инструкция `w.lpszMenuName = NULL` устанавливает имя меню класса окна, т.к. в данной программе меню отсутствует то значение поля установлено в NULL.

`RegisterClass(&w)` — регистрируем класс окна с помощью функции `RegisterClass`, в которую передается указатель на структуру типа `WNDCLASS`.

Далее создаем окно. Это производится с помощью функции `CreateWindow`.

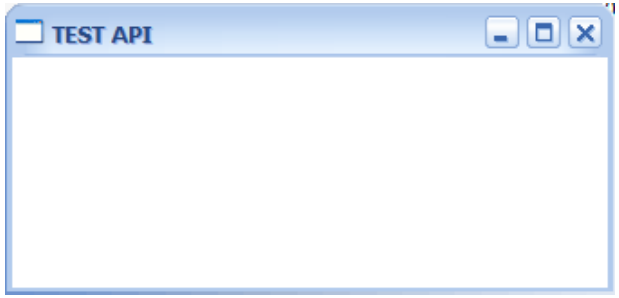
```
hwnd = CreateWindow(
"API WINDOW",
"TEST API",
WS_OVERLAPPEDWINDOW,
10, 10, 600, 480, NULL, NULL,
hInstance, NULL);
```

Разберем параметры функции более детально:

Первый параметр функции (**"API WINDOW"**) является именем зарегистрированного класса окна. Таким образом этот параметр связывает окно с классом окна.

Второй параметр (**"TEST API"**) устанавливает текст заголовка окна.

Параметр **WS_OVERLAPPEDWINDOW** определяет стиль окна. Использование этой константы подразумевает создание стандартного окна *ОС Windows*, т.е. с заголовком, системным меню слева на строке заголовка, иконками для сворачивания, разворачивания и закрытия окна справа на строке заголовка и рамкой окна.



Данный параметр имеет множество вариаций, например определены следующие стили окон: **WS_BORDER**, **WS_CAPTION**, **WS_CHILD**, **WS_CHILDWINDOW**, **WS_CLIPCHILDREN**, **WS_CLIPSIBLINGS**, **WS_DISABLED**, **WS_DLGFRAME**, **WS_GROUP**, **WS_HSCROLL**, **WS_ICONIC**, **WS_MAXIMIZE** и др.

Следующие два параметра (`10, 10`) устанавливают координаты левого верхнего угла окна, соответственно по осям X и Y. Возможно также установить параметр `CW_USEDEFAULT`, тогда окно будет создано в левом верхнем углу экрана.

Параметры `600` и `480` отвечают за ширину и высоту окна соответственно. Если использовать параметр `CW_USEDEFAULT`, то *Windows* создаст окно с размерами по умолчанию.

Следующий параметр описывает *дескриптор* «родительского окна». Так как в данной программе лишь одно окно, то значение установлено в NULL.

Следующий параметр описывает *дескриптор* меню, но так как меню отсутствует то значение установлено в NULL.

Следующий параметр (`hInstance`) описывает *дескриптор* того программного модуля, который создает окно.

Последний параметр функции содержит указатель на дополнительные параметры, однако в данной программе они не используются, поэтому значение установлено в NULL.

Вызов **CreateWindow** возвращает *дескриптор* созданного окна. Этот *дескриптор* хранится в переменной ***hwnd***, которая имеет тип **HWND**.

В общем виде функция **CreateWindow** имеет вид:

```
HWND CreateWindow   (
     LPCTSTR lpClassName,
     LPCTSTR lpWindowName,
     DWORD dwStyle,
     int x,
     int y,
     int nWidth,
     int nHeight,
     HWND hWndParent,
     HMENU hMenu,
     HINSTANCE hInstance,
     LPVOID lpParam
);
```

К тому времени, когда функция **CreateWindow** возвращает управление программе, окно уже создано внутри *Windows*. Однако, на экране монитора оно еще не появилось. Чтобы это осуществить, необходимы следующие функции:

ShowWindow(hwnd,nCmdShow) — данная функция отображает окно на экране. В качестве параметров выступают дескриптор созданного окна и параметр который задает начальный вид окна на экране, так как мы его не определили, то будет использоваться параметр по умолчанию **SW_SHOWDEFAULT**. Возможны также параметры **SW_SHOWNORMAL**, **SW_SHOWMINNOACTIVE**.

UpdateWindow(hwnd) — данная функция осуществляет перерисовку рабочей области окна. Для этого в оконную процедуру (**WndProc**) посылается сообщение **WM_PAINT**. После вызова функции ***UpdateWindow***, окно окончательно выводится на экран.

Затем начинаем описывать цикл обработки сообщений.

```
    while(GetMessage(&msg,NULL,0,0))
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
    return msg.wParam;
```

ОС Windows поддерживает «очередь сообщений» для каждой программы, работающей в данный момент в системе *Windows*. Когда происходит ввод информации, *Windows* преобразует ее в «сообщение», которое помещается в очередь сообщений программы. Затем программа извлекает сообщения из очереди сообщений, выполняя блок команд.

Функция **GetMessage(&msg,NULL,0,0)** извлекает сообщение из очереди сообщений. В качестве параметров этой функции выступают:

Указатель на структуру сообщения **msg** типа **MSG**. Второй, третий и четвертый параметры, NULL или 0, показывают, что программа получает все сообщения от всех окон, созданных этой программой.

Если поле **message** сообщения, извлеченного из очереди сообщений, равно любому значению, кроме **WM_QUIT**, то функция **GetMessage** возвращает ненулевое значение. Сообщение **WM_QUIT** заставляет программу прервать цикл обработки сообщений.

Функция **TranslateMessage(&msg)** передает структуру **msg** обратно в *Windows* для преобразования какого-либо сообщения с клавиатуры.

Функция **DispatchMessage(&msg)** также передает структуру **msg** обратно в *Windows*. *Windows* отправляет сообщение для его обработки соответствующей оконной процедуре — таким образом, *Windows* вызывает оконную процедуру (**WndProc**).

После того, как **WndProc** обработает сообщение, оно возвращается в *Windows*, которая все еще обслуживает вызов функции **DispatchMessage**. Когда *Windows* возвращает управление в программу к следующему за вызовом **DispatchMessage** коду, цикл обработки сообщений в очередной раз возобновляет работу, вызывая **GetMessage**.

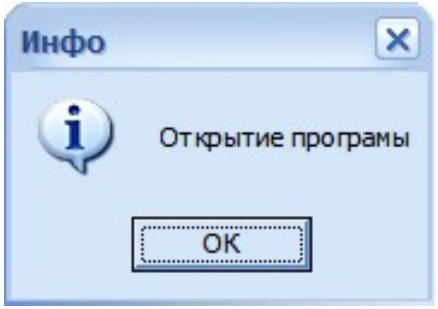
Далее описываем функцию обработки сообщений:

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT Message, WPARAM wparam,LPARAM lparam)
{
   if ( Message == WM_LBUTTONUP )
   {     Beep(39,100);
   }
   if ( Message == WM_DESTROY )
   {     PostQuitMessage(0);
         return 0;
   }
   if ( Message == WM_CLOSE )
   {     MessageBox(NULL,   "Завершение   программы",   "Инфо",   MB_OK   |
   MB_ICONINFORMATION);
   }
```

```
    if ( Message == WM_CREATE )
    {    MessageBox(NULL, "Открытие программы","Инфо",MB_OK |
    MB_ICONINFORMATION);
    }    return DefWindowProc(hwnd, Message, wparam, lparam);
}
```

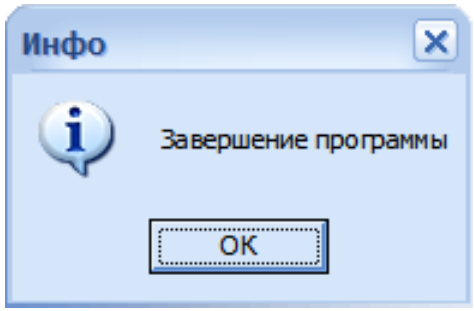
Из кода видно, что **WndProc** обрабатывает четыре сообщения: **WM_LBUTTONUP**, **WM_DESTROY**, **WM_CLOSE**, **WM_CREATE**.

Самым первым обрабатывается сообщение **WM_CREATE**. **WndProc** получает это сообщение тогда, когда *Windows* обрабатывает функцию **CreateWindow** в **WinMain**, то есть когда создается главное окно программы. При создании окна программы на экран выведется сообщение, которое определяется функцией **MessageBox(NULL,"Открытие программы","Инфо",MB_OK | MB_ICONINFORMATION)**.



Сообщение **WM_LBUTTONUP** обрабатывается, если нажата левая кнопка мыши в пределах окна программы. Реакцией на это действие будет «гудок» системного динамика. «Гудок определяется» функцией **Beep()**.

При закрытии программы посылается сообщение **WM_CLOSE**, обработчик которого выведет на экран окно с помощью функции **MessageBox** и при нажатии на кнопку OK приложение будет закрыто.



Еще одним важным сообщением является сообщение **WM_DESTROY**. Это сообщение показывает, что *Windows* находится в процессе ликвидации окна в ответ на полученную от пользователя команду. Программа стандартно реагирует на это сообщение, вызывая:

```
PostQuitMessage(0);
```

Эта функция ставит сообщение **WM_QUIT** в очередь сообщений программы. Функция **GetMessage** возвращает ненулевое значение при любом сообщении, полученном из очереди сообщений за исключением **WM_QUIT**. Когда **GetMessage** получает сообщение **WM_QUIT**, функция возвращает 0. Это заставляет **WinMain** прервать цикл обработки сообщений и выйти в систему, закончив программу.

Сообщения, не обрабатываемые оконной процедурой, должны передаваться функции *Windows*, которая называется **DefWindowProc**. Вызов этой функции приводит к обработке по умолчанию сообщения, которое не обрабатывает оконная процедура. гарантируя таким образом, что все посылаемые приложению сообщения будут корректно обработаны.

Теперь разберем последовательность создания приложения под *ОС Windows* с использованием библиотеки *MFC*.

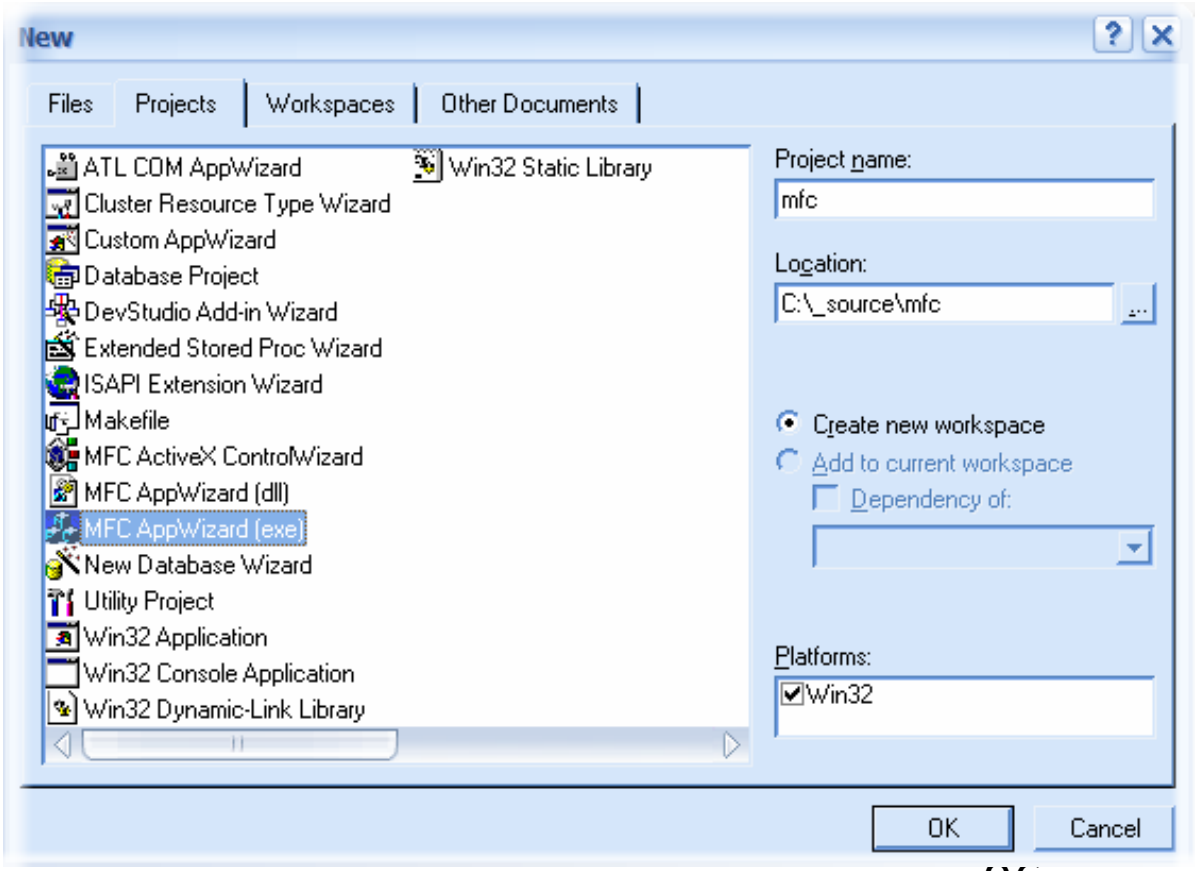
Создаваемое приложение состоит из главного окна и размещенных на нем элементов управления — кнопок, окна списка, поля ввода и ниспадающего списка. Также присутствует модальное окно со сведениями о программе.

Создаваемая программа должна выполнять следующее:

1. Заносить в ниспадающий список (*Combo Box*) и окно списка (*List Box*) данные из поля ввода, при нажатии кнопки «*Добавить*»
2. Удалять выбранный элемент из окна списка при нажатии кнопки «*Удалить*»
3. При нажатии кнопки «*О программе…*» должно появляться модальное окно
4. При нажатии кнопки «*Выход*» приложение должно закрываться.

При создании приложения будем использовать *MFC AppWizard*.

Для этого запускаем среду разработки *Microsoft Visual C++*, в меню *File* выбираем *New*. В появившемся окне на вкладке Projects выбираем *MFC AppWizard (exe)*, далее указываем рабочий каталог проекта и его название.

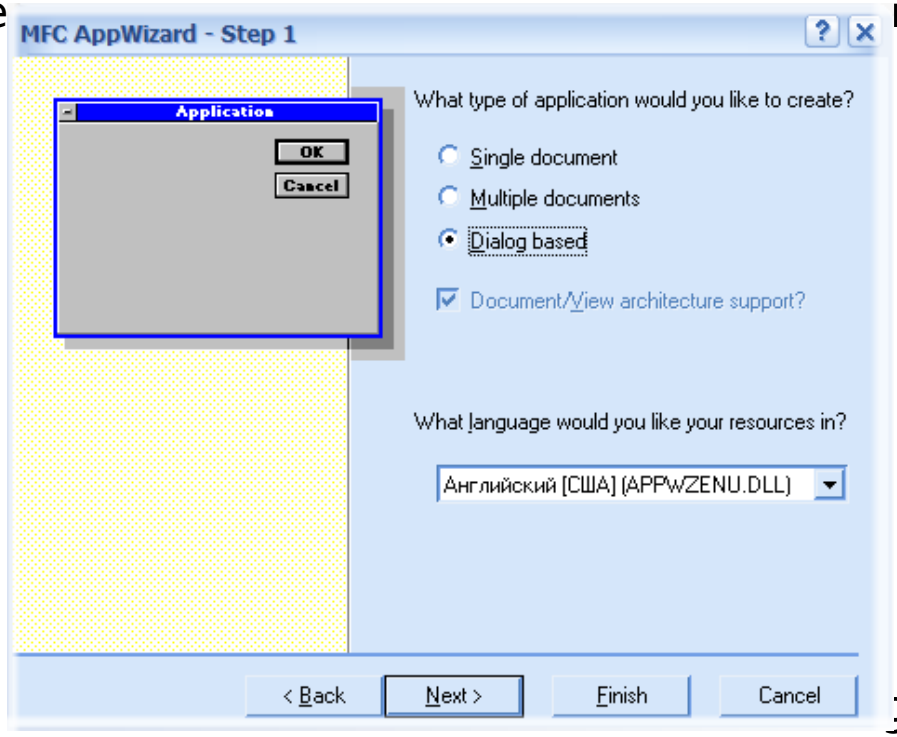


В следующем диалоговом окне нужно выбрать тип создаваемого приложения.

Мастер AppWizard позволяет создавать следующие типы приложений:

1. *Single Document* — однооконное приложение
2. *Multiple Document* — многооконное приложение
3. *Dialog Based* — диалоговое приложение.

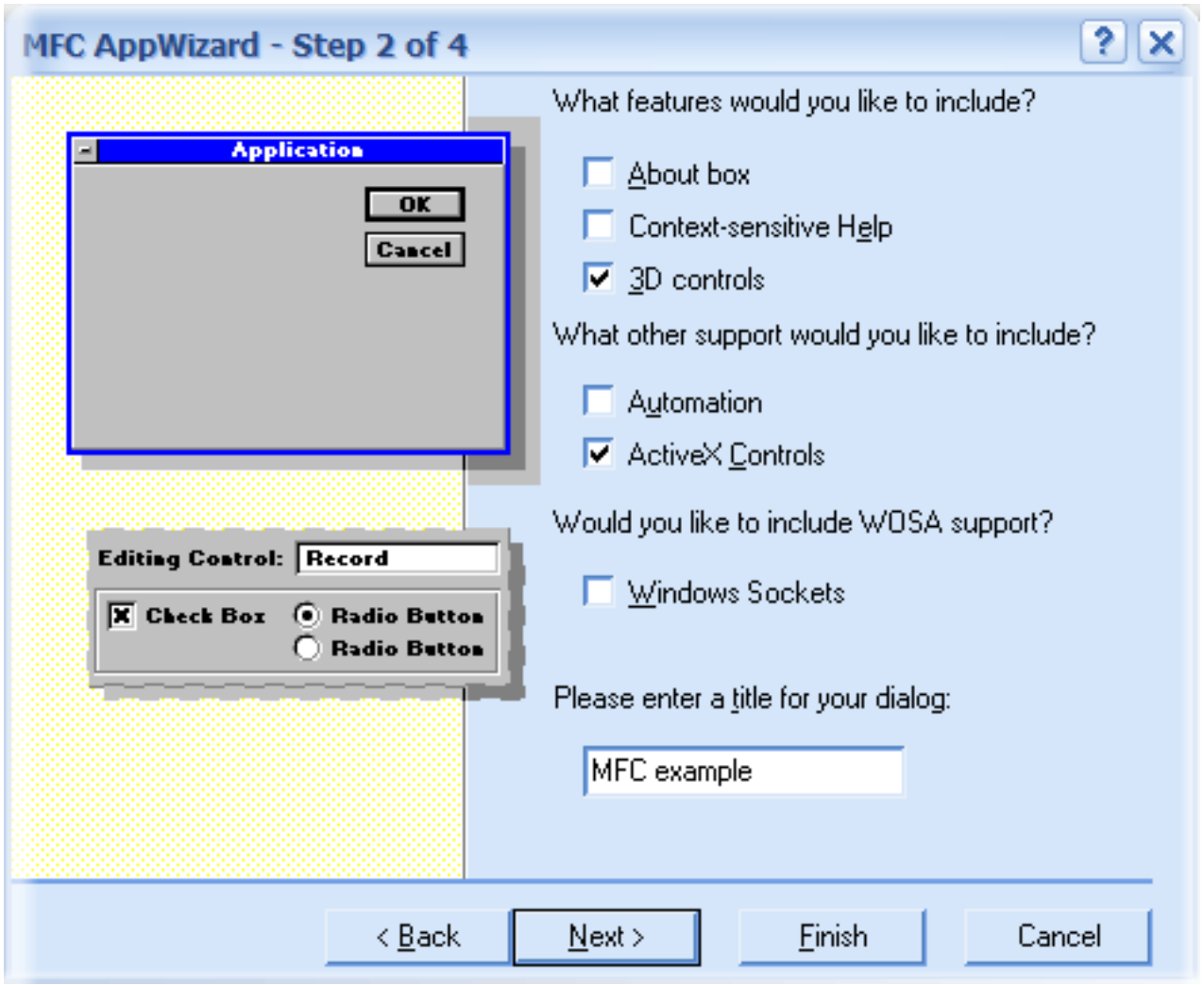
Создаваемый проект будет иметь тип *Dialog Based*, поэтому нужно установить пере ние:



При нажатии на кнопку *Next >* появится диалоговое окно, в котором предлагается выбрать дополнительные возможности создаваемого приложения такие как:

1. *About box* — определяет возможность вызова меню из создаваемого диалогового окна, в котором будет содержаться информация о версии программы.
2. *Context-sensitive Help* — определяет возможность включения в создаваемое приложение справочной системы
3. *3D Controls* — определяет внешний вид элементов управления диалогового окна
4. *Automation* — определяет способность приложения отдать управление другому приложению через механизм *Automation*
5. *ActiveX Controls* — определяет возможность использования в создаваемом приложении элементов *ActiveX* для обмена данными между приложениями
6. *Windows Sockets* — определяет возможность приложения работать с *TCP/IP* сетями
7. В текстовом поле *Please enter a title for your dialog* указывается заголовок диалогового окна.

В рассматриваемом приложении нужно установить опции, как показано на рисунке:



В следующем диалоговом окне мастера предлагается установить настройки по генерации заготовки программы, такие как:

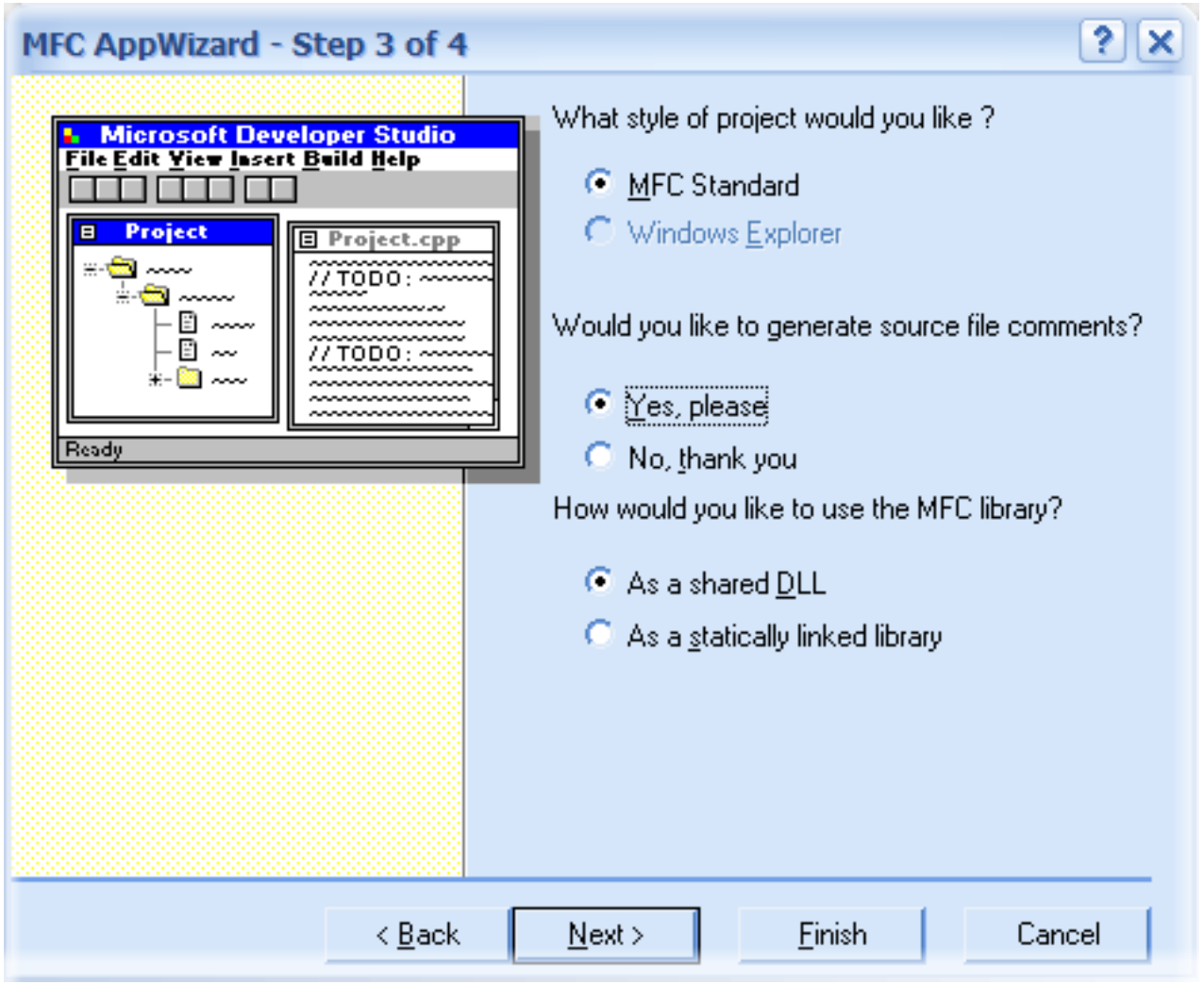
1. *MFC Standard* — стандартное приложение *MFC*
2. *Windows Explorer* — стиль браузера *Internet* (возможность переключения в это положение в данном случае отсутствует)
3. Группа переключателей *Would you like to generate source file comments?* (*Включать ли комментарии в тексты исходных файлов?*) имеет следующие положения:

   Yes, please — да и *No, thank you* — нет
4. Группа переключателей *How would you like to use the MFC library?* (*В каком виде использовать библиотеку MFC?*) имеет следующие положения:

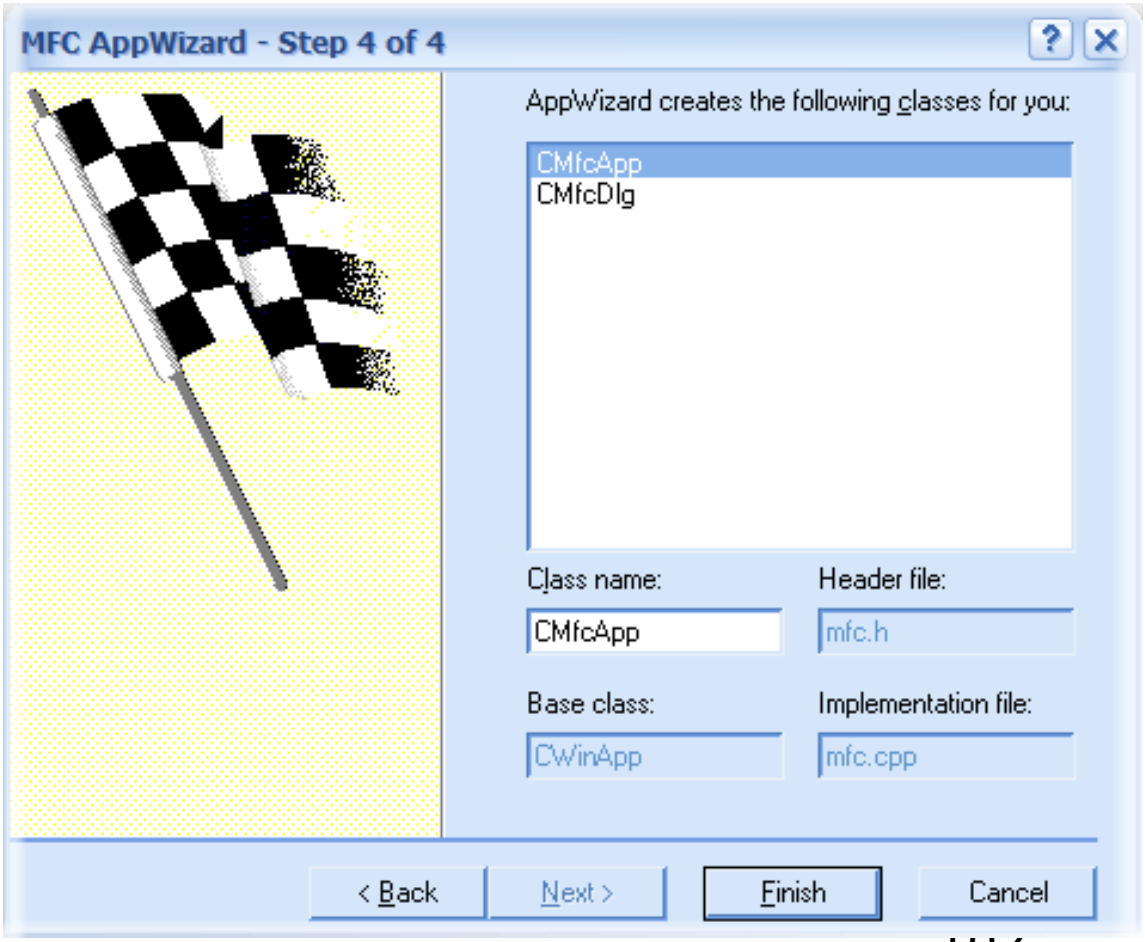
   As a shared DLL — как разделяемые библиотеки динамической компоновки ; такой выбор уменьшает размер исполняемого файла и снижает объем используемой оперативной памяти

   As a statically linked library — как статические библиотеки; такой выбор позволяет программе работать на компьютере, где не установлены необходимые *DLL-библиотеки*.

В данном диалоговом окне ничего менять ненужно, все настройки остаются по умолчанию:

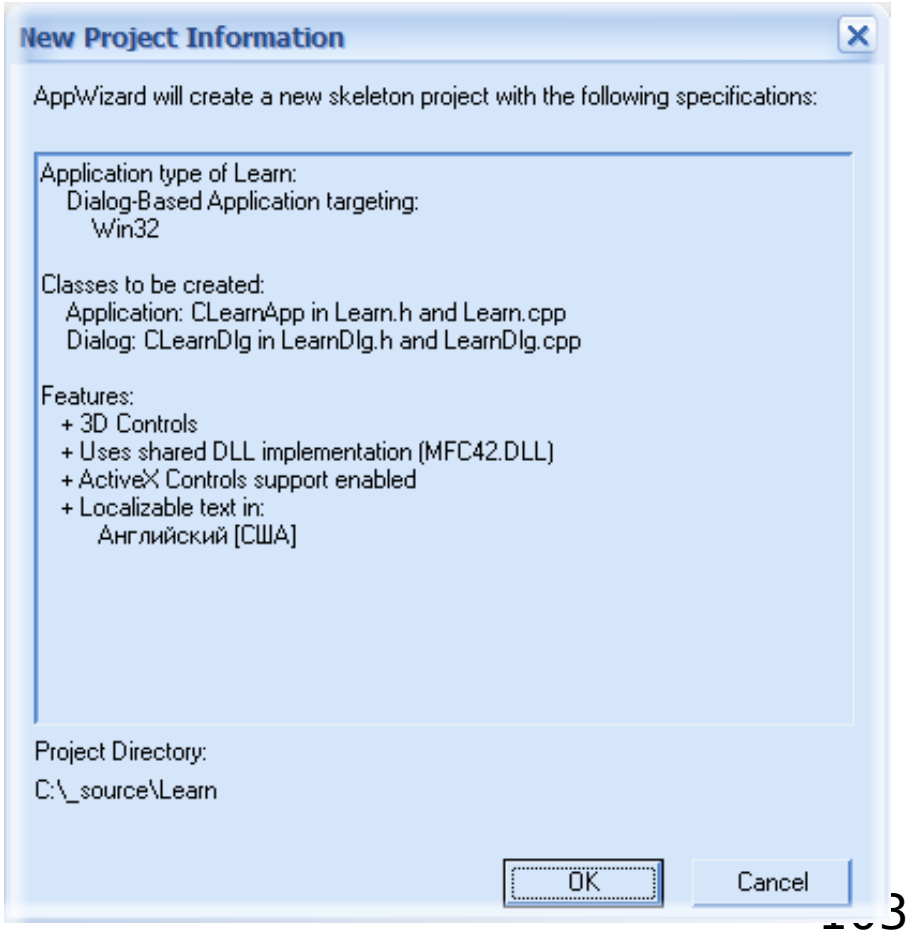


На следующем этапе перечисляются создаваемые мастером *AppWizard* классы. При выделении класса в верхнем окне в соответствующих текстовых полях, расположенных ниже, появляется имя этого класса, имя базового класса, имя файла заголовка, в котором содержится заголовок данного класса, и имя файла реализации, в котором содержится описание методов данного класса.

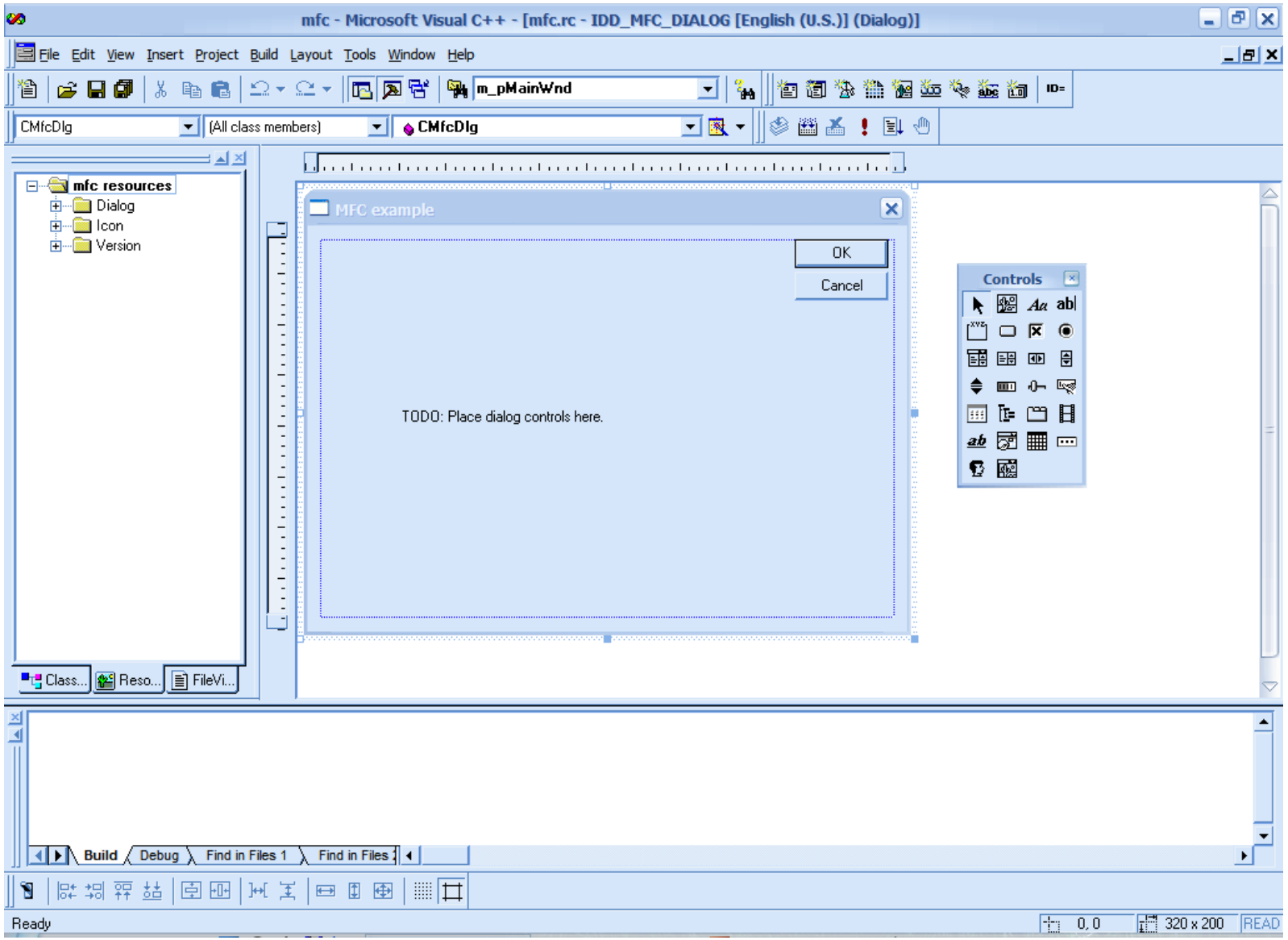


Это окно является последним окном мастера приложений. В нем ничего менять ненужно.

Все опции проекта установлены и при нажатии на кнопку *Finish* появится диалоговое окно *New Project Information*, в котором будут перечислены значения всех ранее установленных параметров:

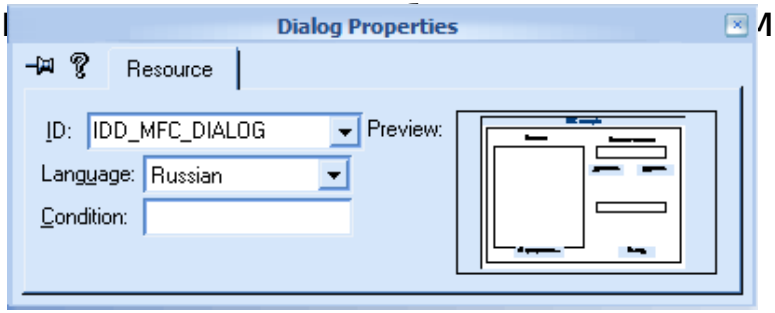


При нажатии на кнопку *OK* система генерирует необходимые файлы и открывает заготовку диалогового окна:



Далее начинаем создавать интерфейс приложения.

Сначала на вкладке *Resource View* окна *Workspace* раскрываем папку *Dialog* и щелкаем правой кнопкой мыши на идентификаторе ресурса *IDD_MFC_DIALOG* и выбираем команду *Properties*. В появившемся окне в списке *Language* выбираем опцию *Russian* — это необходимо для того, чтобы в дальнейшем все надписи на элеме языке.

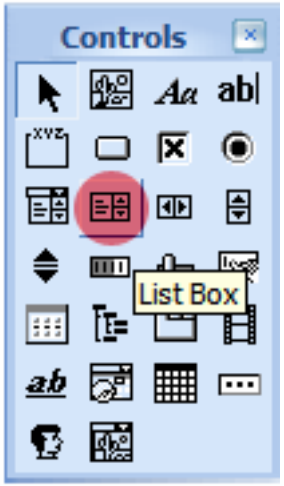


Для доступа к остальным свойствам диалогового окна нужно выбрать его рамку и щелкнуть правой кнопко

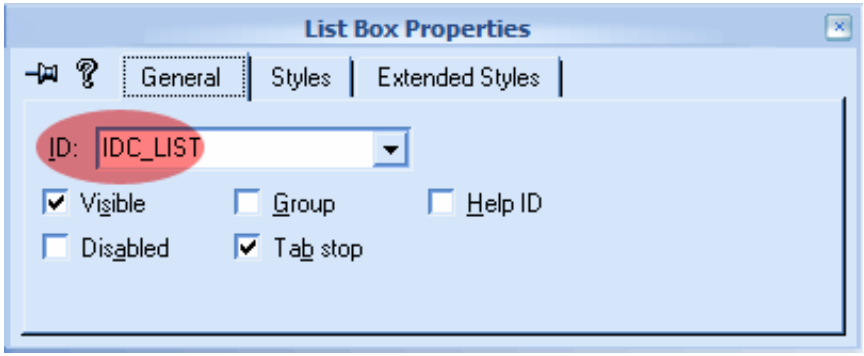
После этого удаляем статический текст «*TODO: Place dialog controls here*» с заготовки диалогового окна и начинаем помещать на него необходимые элементы управления.

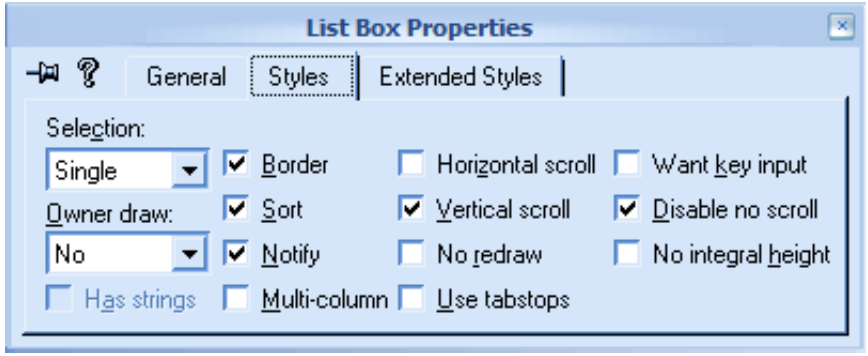
В панели инструментов выбираем элемент управления *List Box* (*Окно списка*)

и помещаем его на заготовку диалогового окна.

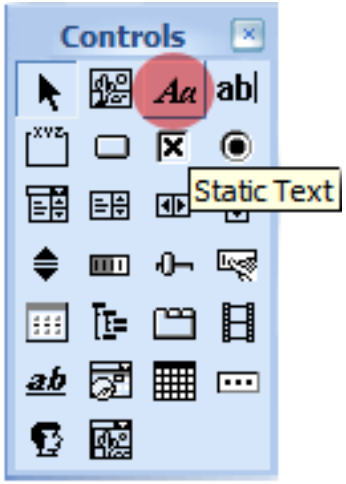


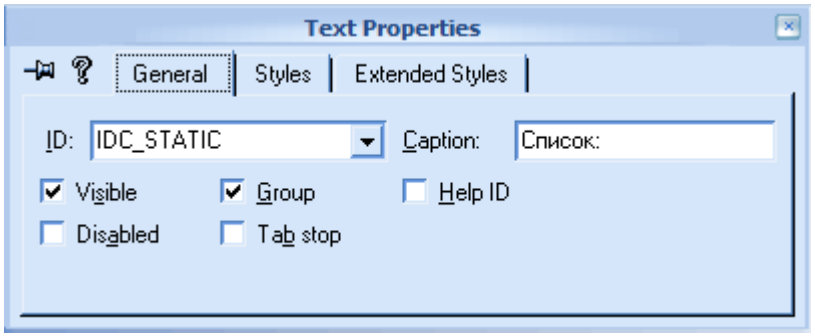
После этого правой кнопкой мыши вызываем контекстное меню у этого элемента, выбираем команду *Properties*. Далее меняем идентификатор ресурса на *IDC_LIST* и устанавливаем следующие опции стиля элемента:



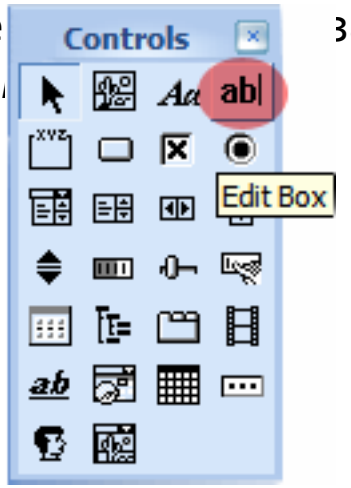


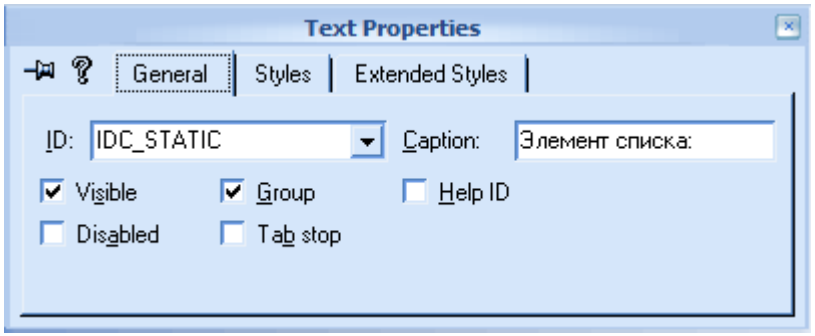
Над компонентом *List Box* поместим статический текст (*Static Text*) «*Список:*»





На панели инструментов выберем элемент управления *Edit Box* и поме заготовку окна. Над ним разместим статический текст «*Эле*



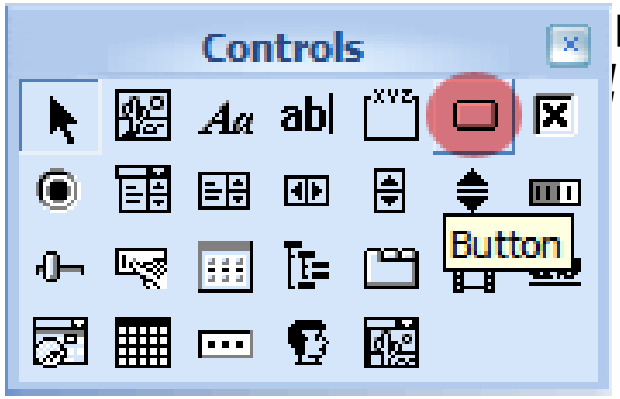


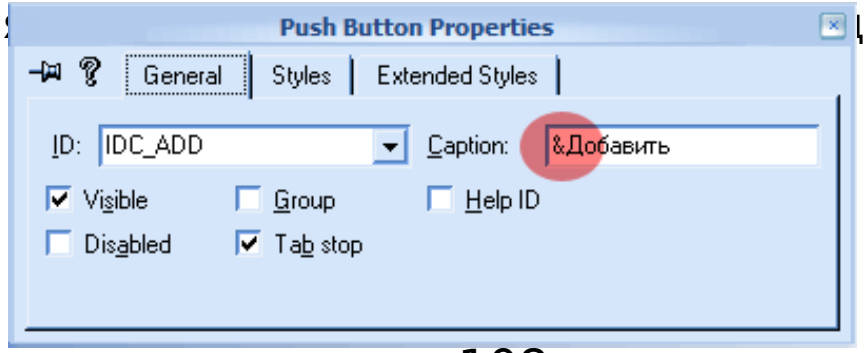
Теперь добавляем кнопки на заготовку окна.

В рассматриваемом приложении присутствует четыре кнопки:

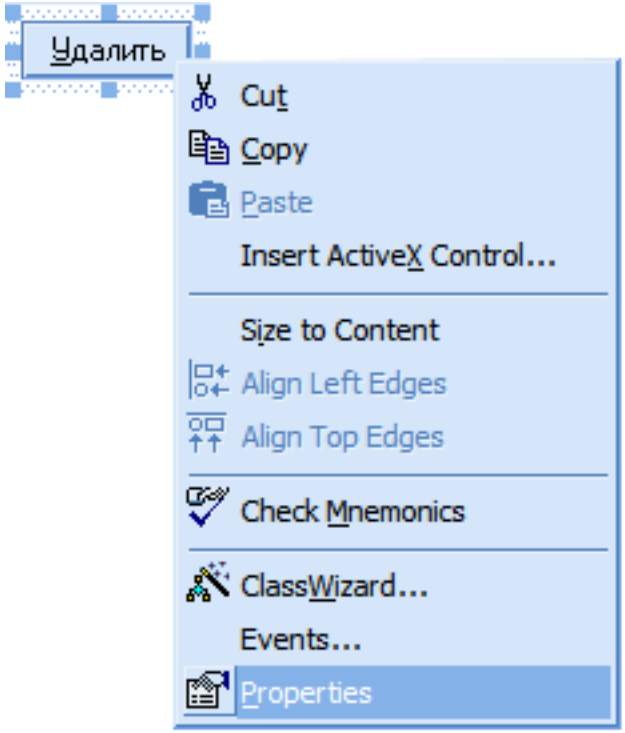
1. *Добавить* — добавление строки из поля ввода (*Edit Box*) в окно списка (*List Box*)
2. *Удалить* — удаление выбранного элемента из окна списка
3. *О программе…* — вызов дочернего окна со сведениями о приложении
4. *Выход* — закрытие программы

В панели инструментов выбираем элемент управления *Button* и перетаскиваем его на форму диалогового окна, а затем устанавливаем свойство *Caption* для каждой кнопки. (Символ & перед буквой «Д» равлени ью /»)

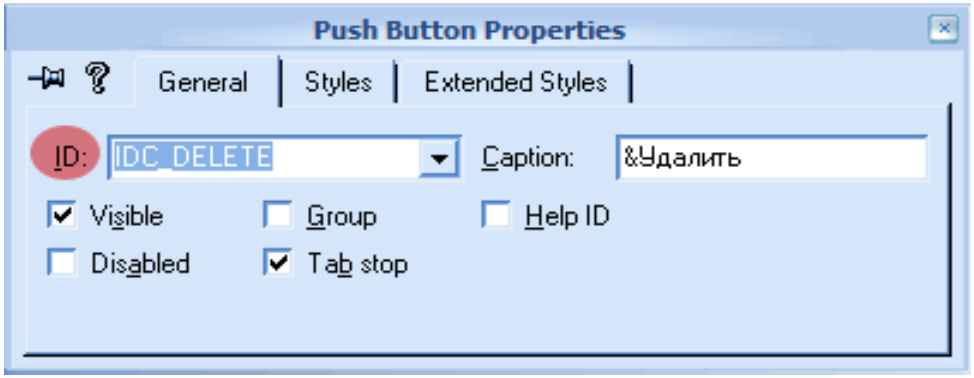




Затем устанавливаем идентификаторы для кнопок. Для этого правой кнопкой мыши вызываем контекстное меню у соответствующей кнопки, затем выбираем команду *Properties* и в текстовом поле *ID:* вводим следующие значения:

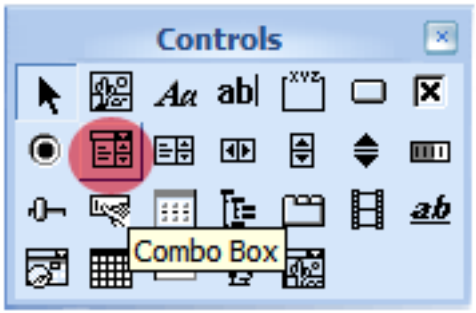


1. Для кнопки *Удалить* — *IDC_DELETE*
2. Для кнопки *Добавить* — *IDC_ADD*
3. Для кнопки *О программе...* — *IDC_ABOUT*
4. Для кнопки *Выход* — *IDC_CLOSE*

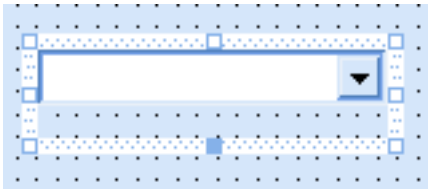


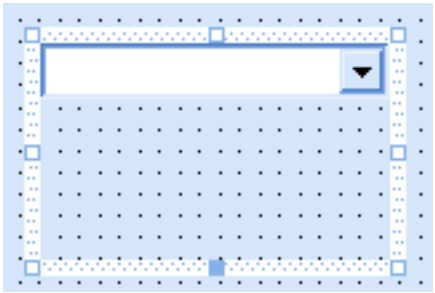
Добавляем на форму элемент управления *Combo Box*. Для этого на панели инструментов выбираем соответствующий элемент и перетаскиваем его на форму.

Идентификатор ресурса оставляем по умолчанию.

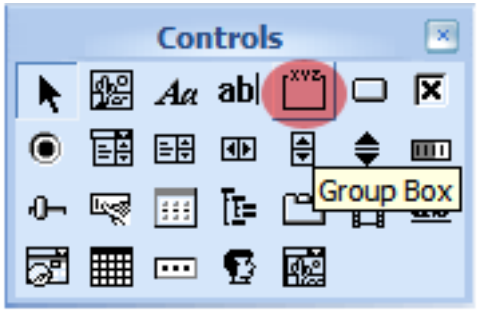


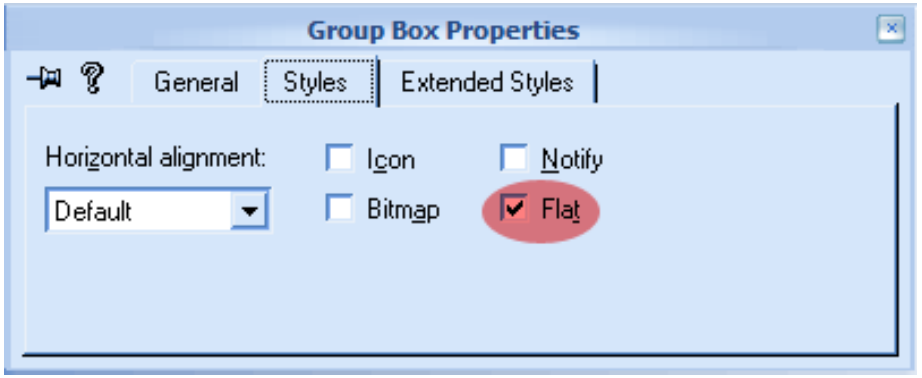
Теперь нужно установить размер ниспадающего списка, чтобы было нагляднее видно элементы, имеющиеся в списке. Для этого щелкаем левой кнопкой мыши по кнопке со стрелкой и растягиваем появившуюся рамку.





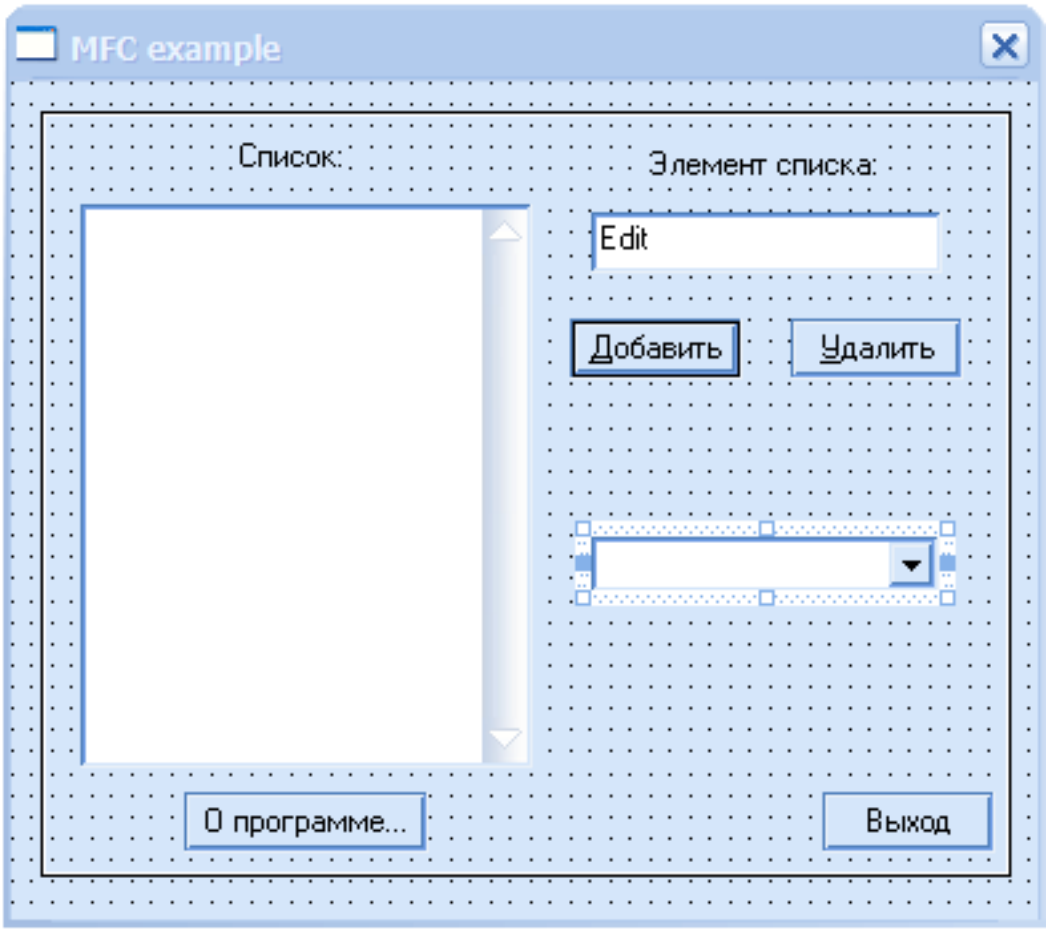
Добавим на форму окна элемент *Group Box*, для этого на панели инструментов выбираем его и перетаскиваем на форму, а затем растягиваем по периметру окна. Затем правой кнопкой мыши вызываем контекстное меню и выбираем команду *Properties*. В появившемся окне свойств стираем содержимое поля Caption: , а на вкладке *Styles* устанавливаем свойство *Flat*



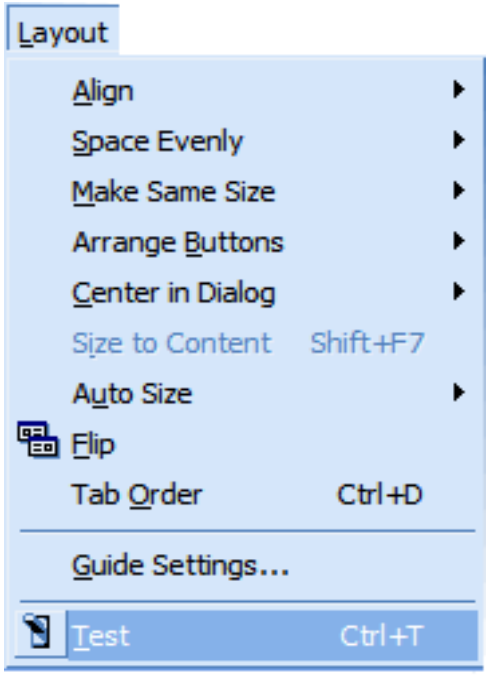


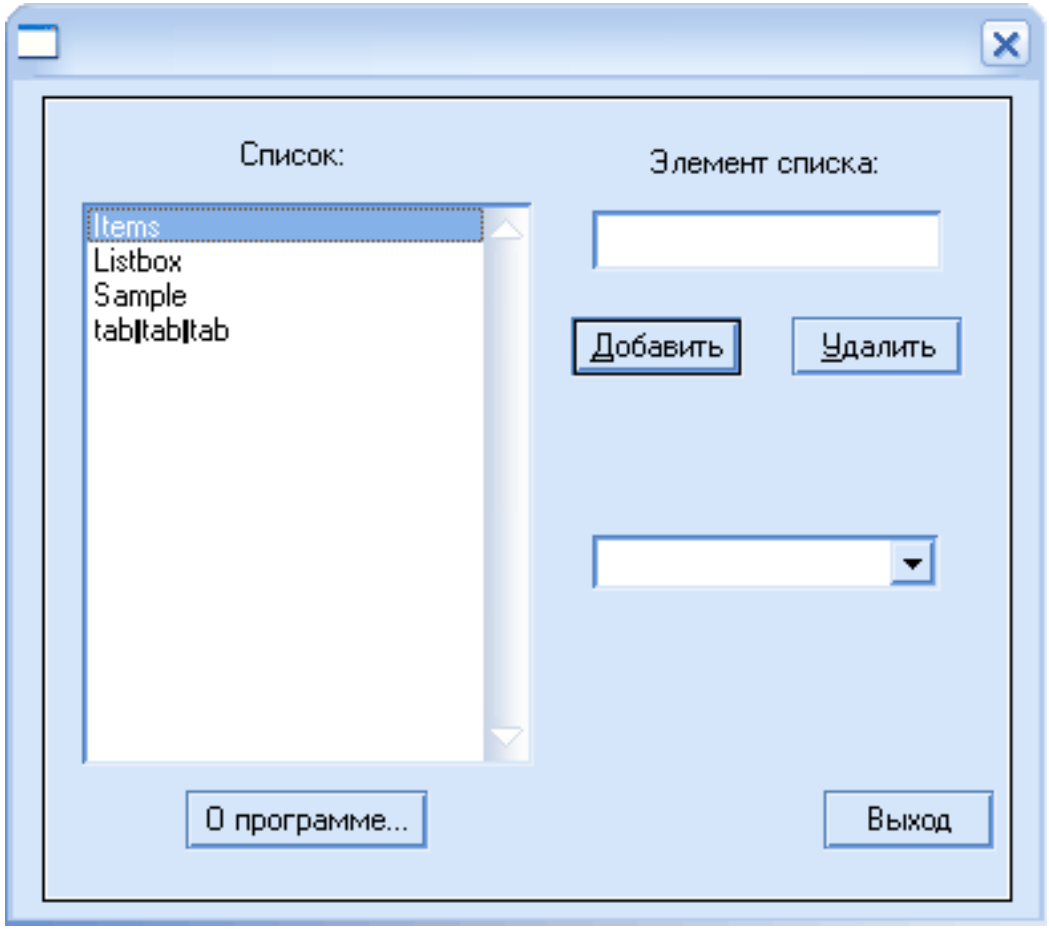
В результате этих действий окно приложения будет иметь рамку.

После всех проделанных операций, заготовка окна приложения примет вид:



Чтобы увидеть как будет выглядеть окно приложения в действительности, иными словами когда приложение будет запущено, нужно выбрать (находясь в режиме редактирования формы) меню *Layout* и дать команду *Test* или можно воспользоваться сочетанием клавиш «*Control + T*»

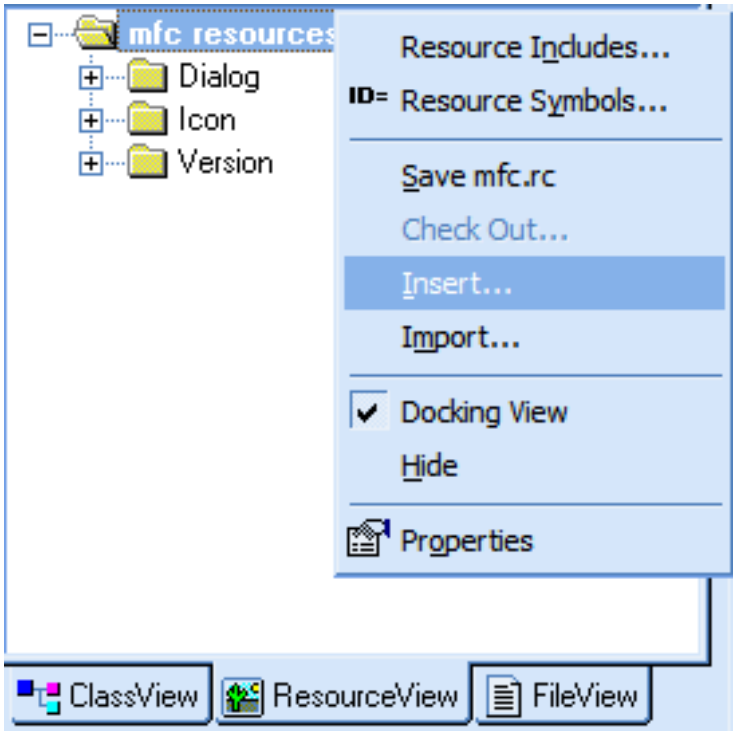


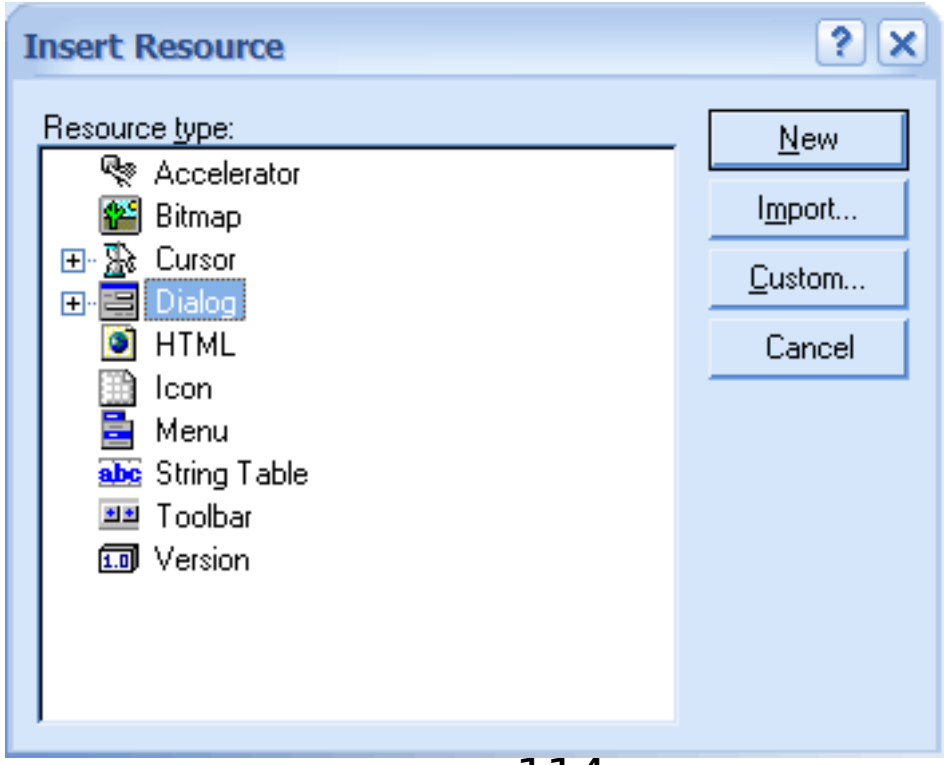


Так как в создаваемом приложении кроме главного окна должно быть окно со сведениями о программе, то его нужно добавить в проект.

Для этого переходим на вкладку *Resources* окна *Project Workspace* и правой кнопкой мыши щелкаем по пункту *mfc resources* выбираем команду *Insert*. В появившемся окне устанавливаем курсор на *Dialog* и нажимаем кнопку *New*.

Для вызова окна *Insert Resource* можно также воспользоваться сочетанием клавиш «*Control* + *R*»



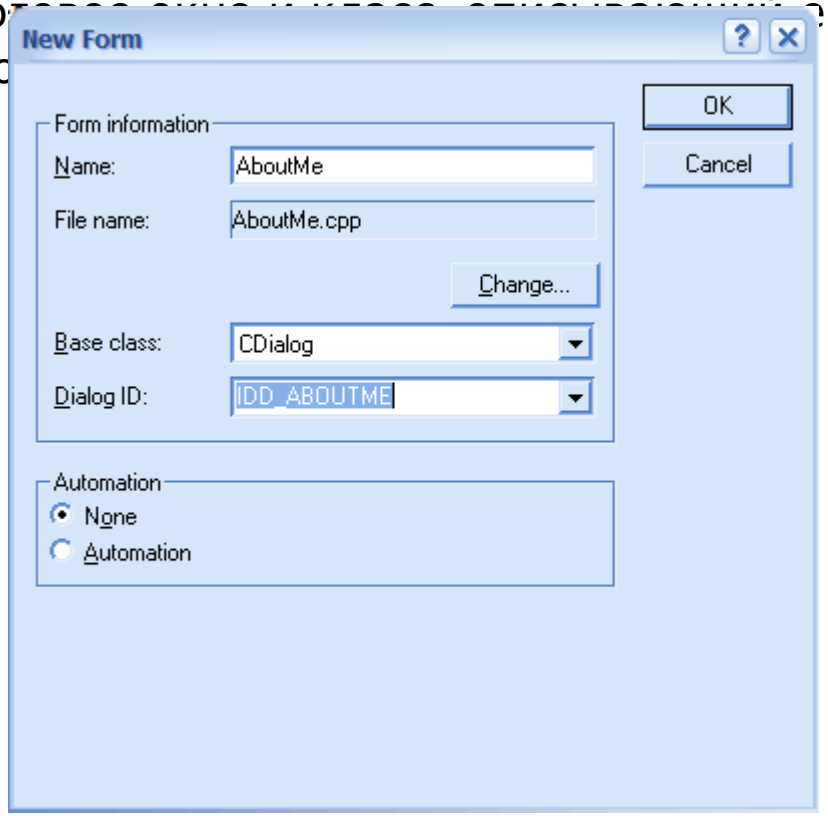


Примечание

Включить в проект новое диалоговое окно можно с помощью меню *Insert*.

Для этого нужно выбрать пункт *New Form* и в появившемся окне заполнить следующие поля: имя формы (имя класса), базовый класс и идентификатор окна.

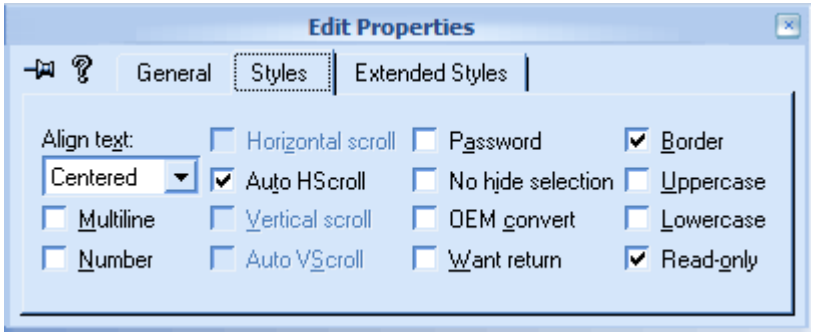
Отличие от предыдущего метода заключается в том, что этим методом мы создаем и диало[illegible]го; в предыдущем же методе создается то[illegible]



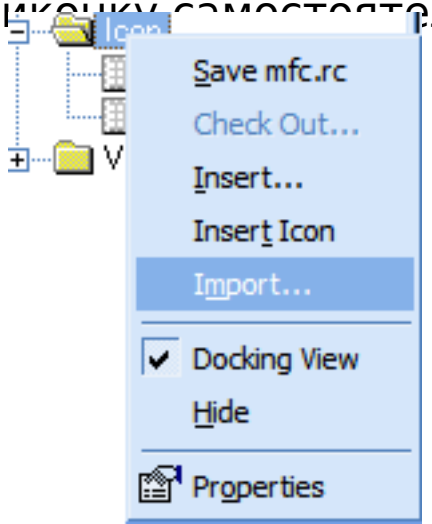
После того как добавлено новое окно, нужно отредактировать его свойства, а именно сменить идентификатор на *IDD_ABOUT* и установить опции отображения окна на экране. Для этого вызываем окно опций диалогового окна и изменяем параметры.

На созданном окне разместим компоненты *Static Text*, *Edit Box* и *Picture*. В поле *Caption* окна свойств статического текста введем текст.

Для компонента *Edit Box* в окне свойств установим следующие параметры:

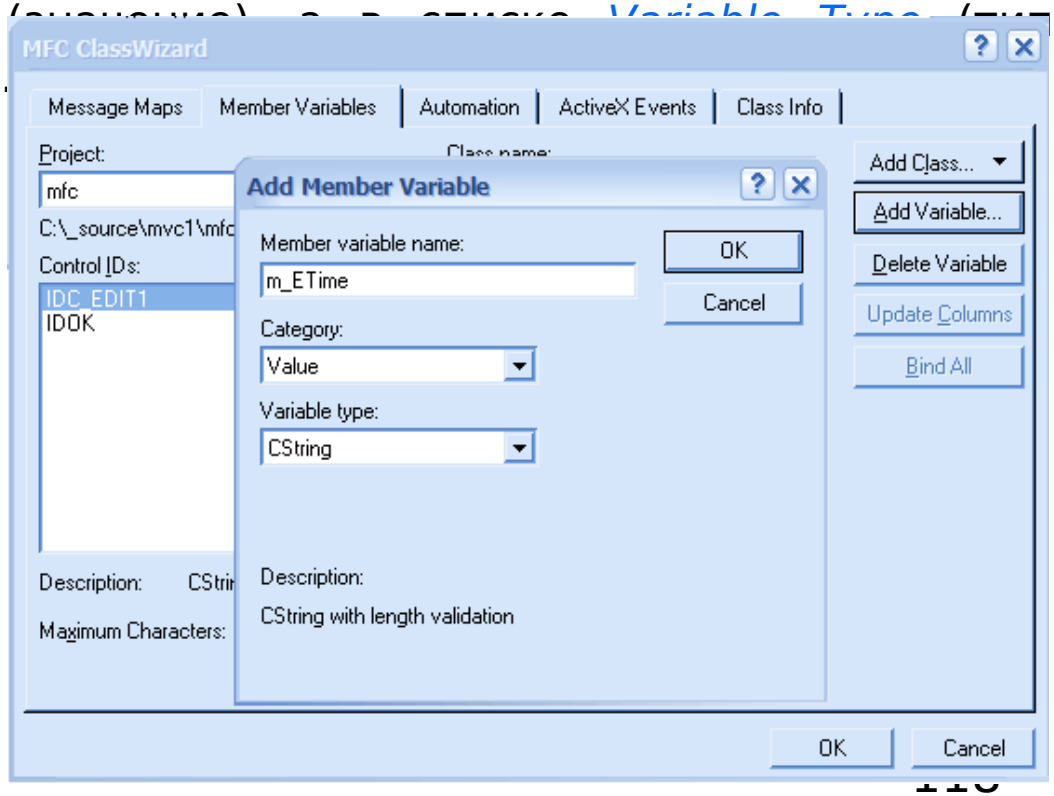


Чтобы добавить иконку, нужно на вкладке *Resources* выбрать элемент *Icon* и щелкнуть правой кнопкой мыши. В появившемся меню выбрать *Import* для загрузки в проект готовой иконки или *Insert Icon* чтобы нарисовать иконку самостоятельно.

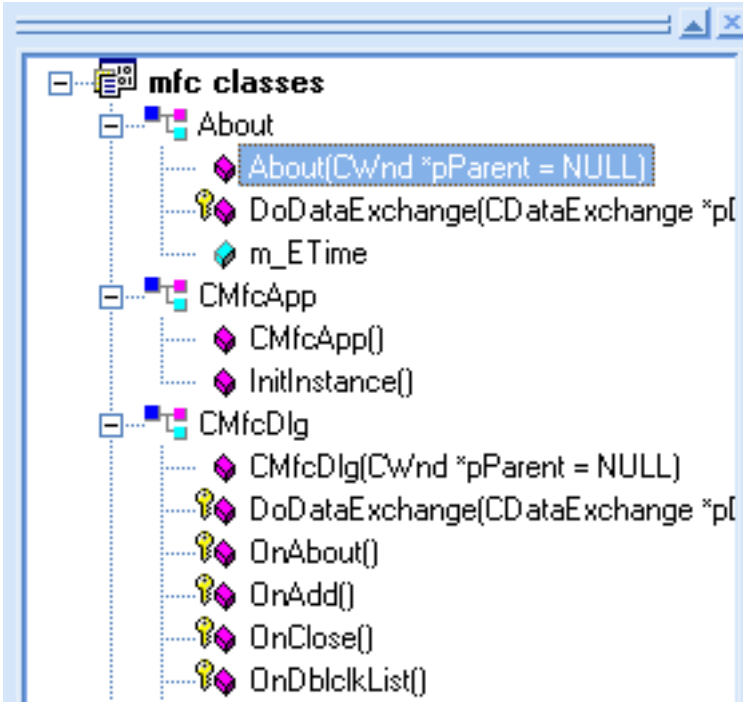


Нам нужно чтобы в поле *Edit Box* выводилась текущая дата, для этого нужно создать переменную для элемента управления.

Вызываем *Class Wizard* (*View* -> *Class Wizard*, «*Control* + *W*») и переходим на вкладку *Member Variables*. В списке Class Name выбираем About, а *в* списке *Object IDs* выбираем идентификатор *IDC_EDIT1* (идентификатор элемента управления *Edit Box*) и нажимаем кнопку *Add Variable*. В появившемся окне *Add Member Variable* в текстовом поле *Member Variable Name* вводим идентификатор *m_ETime*, в списке *Category* выбираем значение *Value* (значение), а в списке *Variable Type* (тип переменной) выбираем *CString*.



Чтобы вывести текущую дату в элемент *Edit Box* без каких либо действий пользователя, нужно написать код в конструкторе диалогового окна. Для этого необходимо открыть вкладку *Class View*, раскрыть класс *About* и два раза щелкнуть левой кнопкой мыши по строке с прототипом конструктора, при этом курсор автоматически переместится в файл реализации.



Измените исходный код конструктора, чтобы он принял следующий вид:

```
About::About(CWnd* pParent /*=NULL*/) : CDialog(About::IDD, pParent)
{
	//{{AFX_DATA_INIT(About)
	m_ETime = _T("");
	//}}AFX_DATA_INIT
	CTime currentTime = CTime::GetCurrentTime();

	CString s = currentTime.Format("%A, %B %d, %#Y");

	m_ETime = s;

}
```

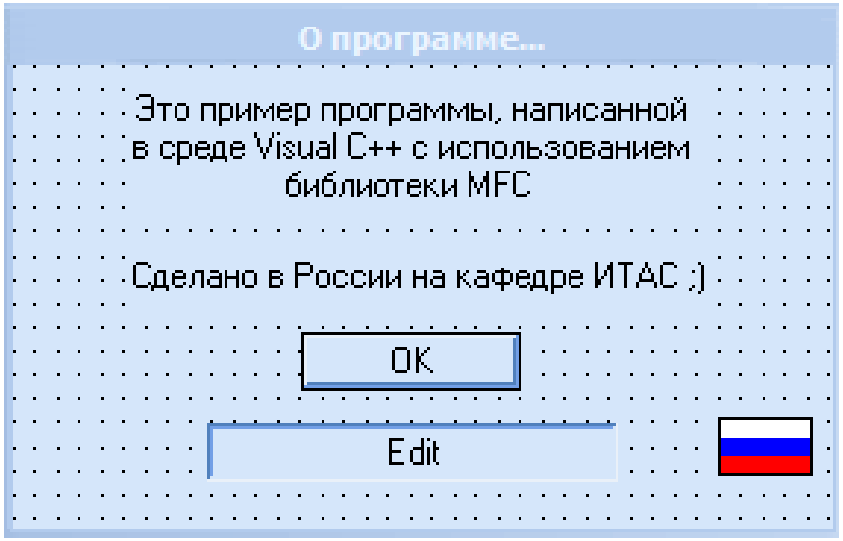
Сначала создаем экземпляр класса *CTime* (класс для работы с датами и временем) с именем **currentTime** и сразу его инициализируем **CTime currentTime = CTime::GetCurrentTime()**. То есть в переменной currentTime будет храниться текущая дата.

Затем создаем экземпляр класса *CString* с именем **s** и сразу его инициализируем **CString s = currentTime.Format("%A, %B %d, %#Y")**. Функция **Format** используется для форматированного вывода времени, ее параметры соответствуют:

%A — день недели, **%B** — месяц, **%d** — день, **%#Y** — год.

m_ETime = s — выводим дату в *Edit Box*.

В итоге получим заготовку окна следующего вида:



Теперь, чтобы работать с элементами управления главного окна, нужно поставить им в соответствие идентификаторы (переменные-члены класса) с помощью которых можно будет оперировать с самим элементом или со значением, находящимся в элементе управления.

Открываем *Class Wizard* (*View -> Class Wizard*, «*Control + W*») и переходим на вкладку *Member Variables*.

В списке *Project* отображаются проекты. Так как мы работаем с одним проектом, то у нас есть только один выбор в данном списке.

В списке *Class Name* находятся классы, доступные в данном приложении.

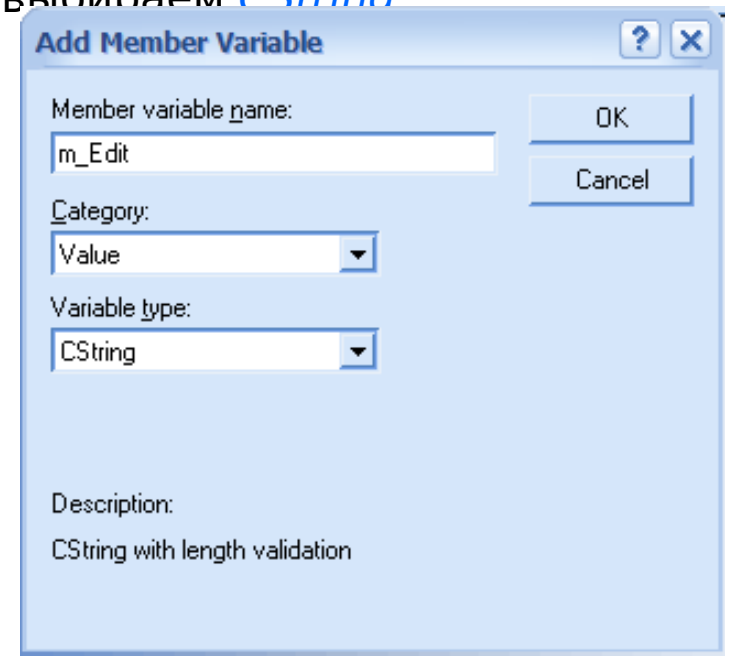
В окне *Control IDs* перечислены идентификаторы ресурсов, доступные в данном классе.

Чтобы добавить переменную для ресурса нужно выбрать соответствующий идентификатор и нажать на кнопку *Add Variable*.

Начинаем добавлять переменные для элементов.

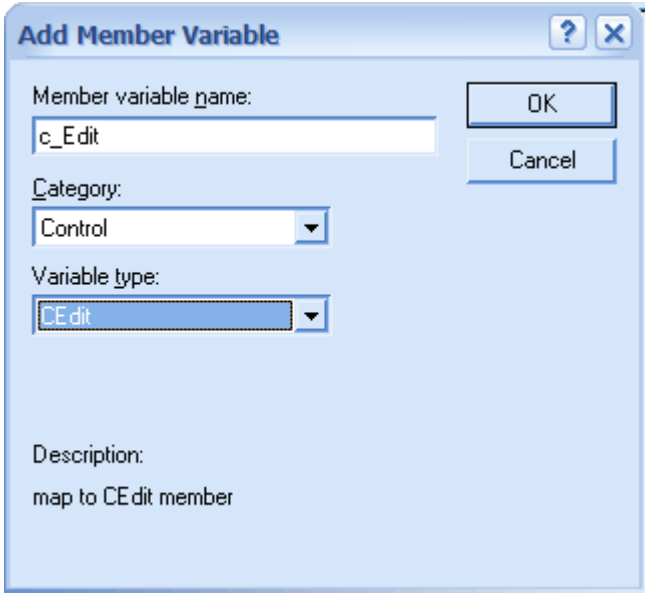
Для элемента управления *Edit Box* добавим две переменные — одну типа *CString* (для работы с содержимым поля ввода), а другую типа *CEdit* (для работы непосредственно с элементом управления):

Вызываем *Class Wizard* (*View* -> *Class Wizard*, «*Control* + *W*») и переходим на вкладку *Member Variables*. В списке *Object IDs* выбираем идентификатор *IDC_EDIT1* (идентификатор элемента управления *Edit Box*) и нажимаем кнопку *Add Variable*. В появившемся окне *Add Member Variable* в текстовом поле *Member Variable Name* вводим идентификатор *m_Edit*, в списке *Category* выбираем значение *Value* (значение), а в списке *Variable Type* (тип переменной) выбираем *CString*.



Аналогично добавляем переменную *c_Edit* для работы с элементом управления. Однако в списке *Category* выбираем значение *Control* (элемент управления). Список *Variable Type* (тип переменной) примет единственное значение *CEdit*.

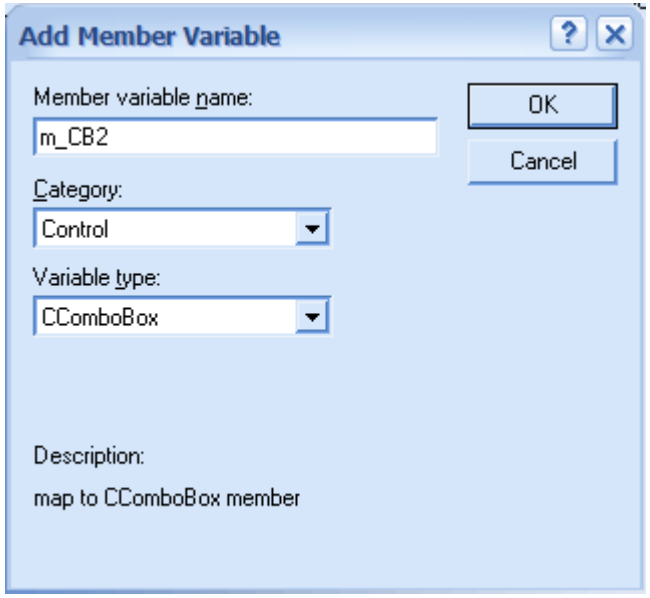
Чтобы при запуске приложения фокус ввода находился на элементе *Edit Box*, нужно добавить строку `c_Edit.SetFocus()` в обработчик сообщения `OnPaint`.

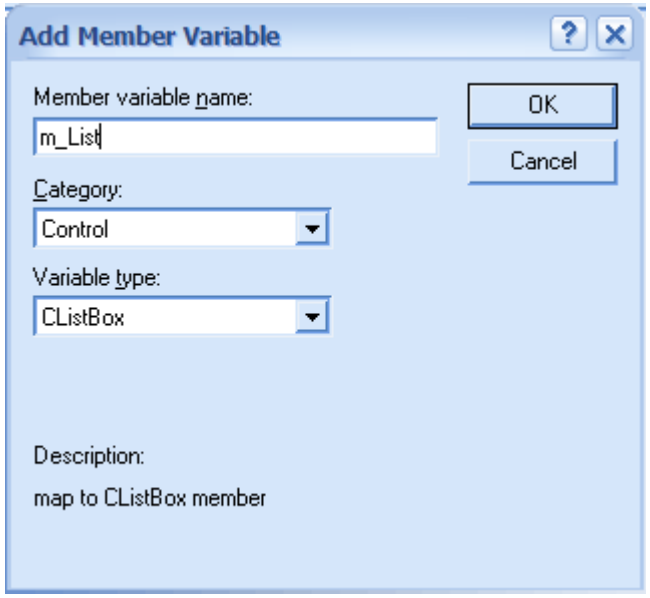


Тем же способом добавляем переменные к остальным элементам управления.

Для элемента *List Box* переменная *m_List* категории *Control* и типа *CListBox*.

Для элемента *Combo Box* переменная *m_CB2* категории *Control* и типа *CComboBox*.





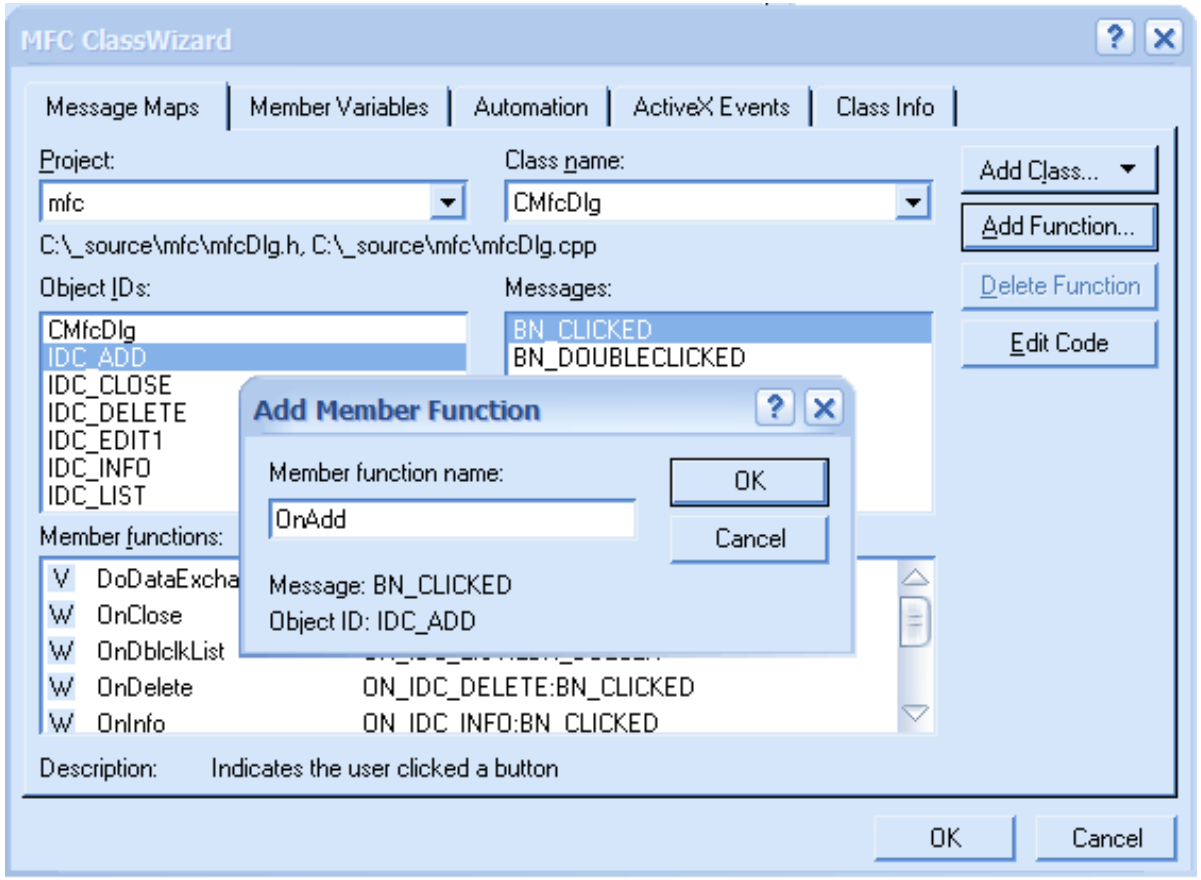
Теперь, когда у необходимых элементов управления есть переменные-идентификаторы, нужно «научить» приложение обрабатывать сообщения.

Для этого следует вызвать *Class Wizard* и перейти на вкладку *Message Maps* (Карты сообщений).

Данная вкладка имеет несколько окон и текстовых полей. В списке *Project* указано имя текущего проекта, но так как мы работаем с одним проектом, то есть лишь один выбор — *mfc*. В списке *Class Name* указываются классы, доступные в проекте. В окне списка *Object IDs* перечислены идентификаторы ресурсов, доступные в классе и имя самого класса, имеющего свой набор сообщений и функций их обработки. В окне списка *Messages* (Сообщения) перечислены все сообщения и функции их обработки, связанные с выделенными в предыдущем окне идентификатором объекта.

Чтобы добавить обработчик события нажатия на кнопку «*Добавить*», нужно проделать следующее:

На вкладке *Message Maps* диалогового окна *MFC Class Wizard* в окне списка *Object IDs* выделить идентификатор кнопки (*IDC_ADD*) и в списке *Messages* выделить сообщение *BN_CLICKED*. Затем нажать кнопку *Add Function*. В появившемся окне ввести имя функции (можно оставить без изменений) и нажать OK.



Если нажать на кнопку *Edit Code*, то курсор автоматически перейдет на только что созданную функция *OnAdd*.

Теперь можно непосредственно начинать программировать обработчик события.

Исходный код обработчика:

```
void CMfcDlg::OnAdd()
{
    UpdateData(TRUE);
if (m_Edit.GetLength() != 0)
{
m_List.AddString(m_Edit);
m_CB2.AddString(m_Edit);
m_Edit = "";
c_Edit.SetFocus();
}
else MessageBox("Нельзя добавить пустую строку!", "Ошибка", MB_OK |
MB_ICONSTOP);

UpdateData(FALSE);
}
```

Обработчик начинается с функции `UpdateData(TRUE)` которая получает информацию из объектов классов элемента управления диалогового окна.

Далее в конструкции `if…else` проверяется корректность ввода данных и их занесение в список.

`if (m_Edit.GetLength() != 0)` — проверка длины введенной строки и если она не равна нулю:

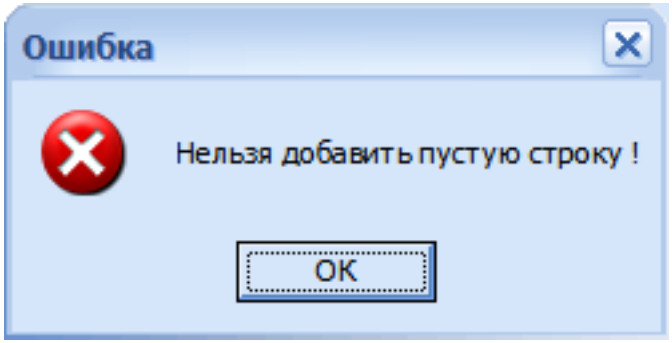
`m_List.AddString(m_Edit)` — добавляем строку из поля ввода в список

`m_CB2.AddString(m_Edit)` — добавляем строку из поля ввода в ниспадающий список

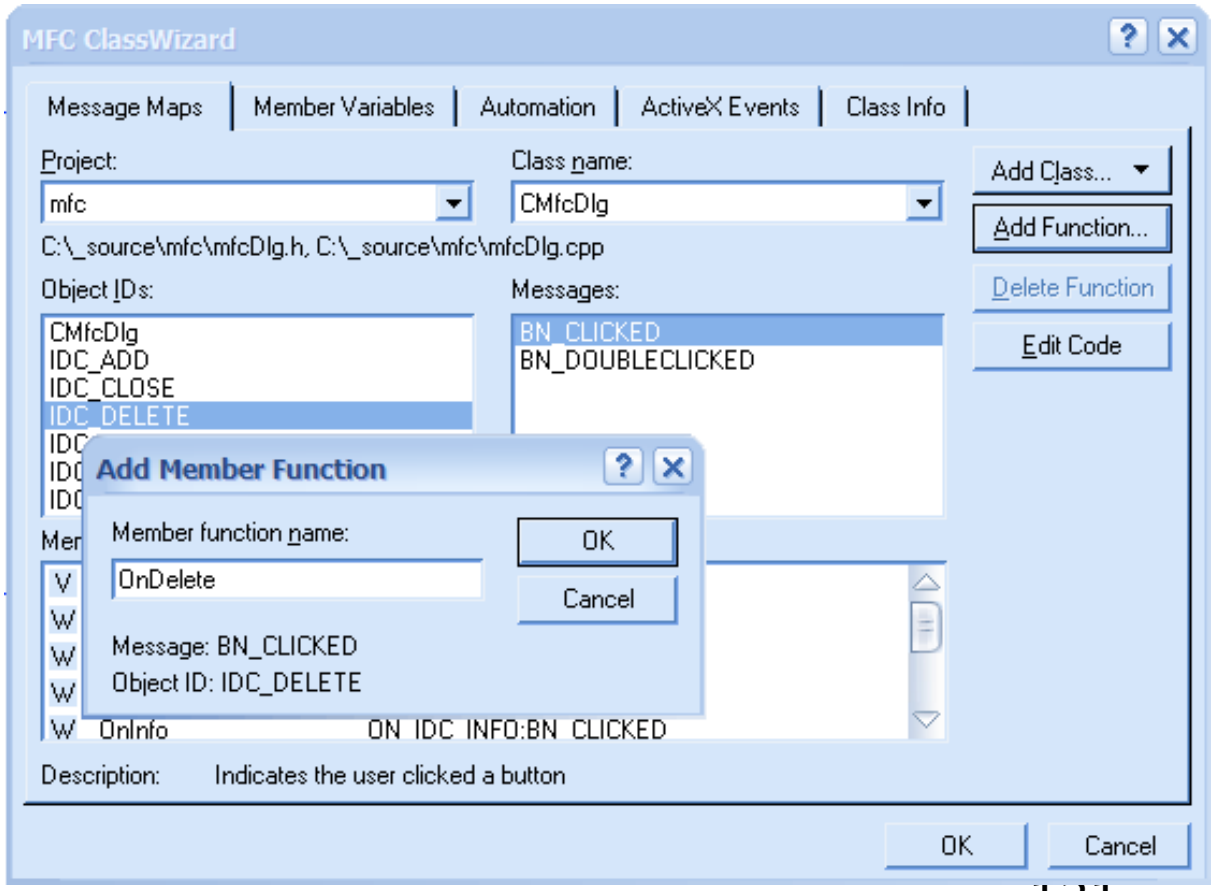
`m_Edit = ""` — очищаем поле ввода

`c_Edit.SetFocus()` — устанавливаем фокус ввода на *Edit Box*

`else MessageBox("Нельзя добавить пустую строку!", "Ошибка", MB_OK | MB_ICONSTOP)` — иначе, если ничего не введено, выводим сообщение об ошибке.



Теперь добавим обработчик события нажатия на кнопку «*Удалить*». Для этого на вкладке *Message Maps* диалогового окна *MFC Class Wizard* в окне списка *Object IDs* нужно выделить идентификатор кнопки (*IDC_DELETE*) и в списке *Messages* выделить сообщение *BN_CLICKED*. Затем нажать кнопку *Add Function*. В появившемся окне ввести имя функции (можно оставить без изменений) и нажать OK.



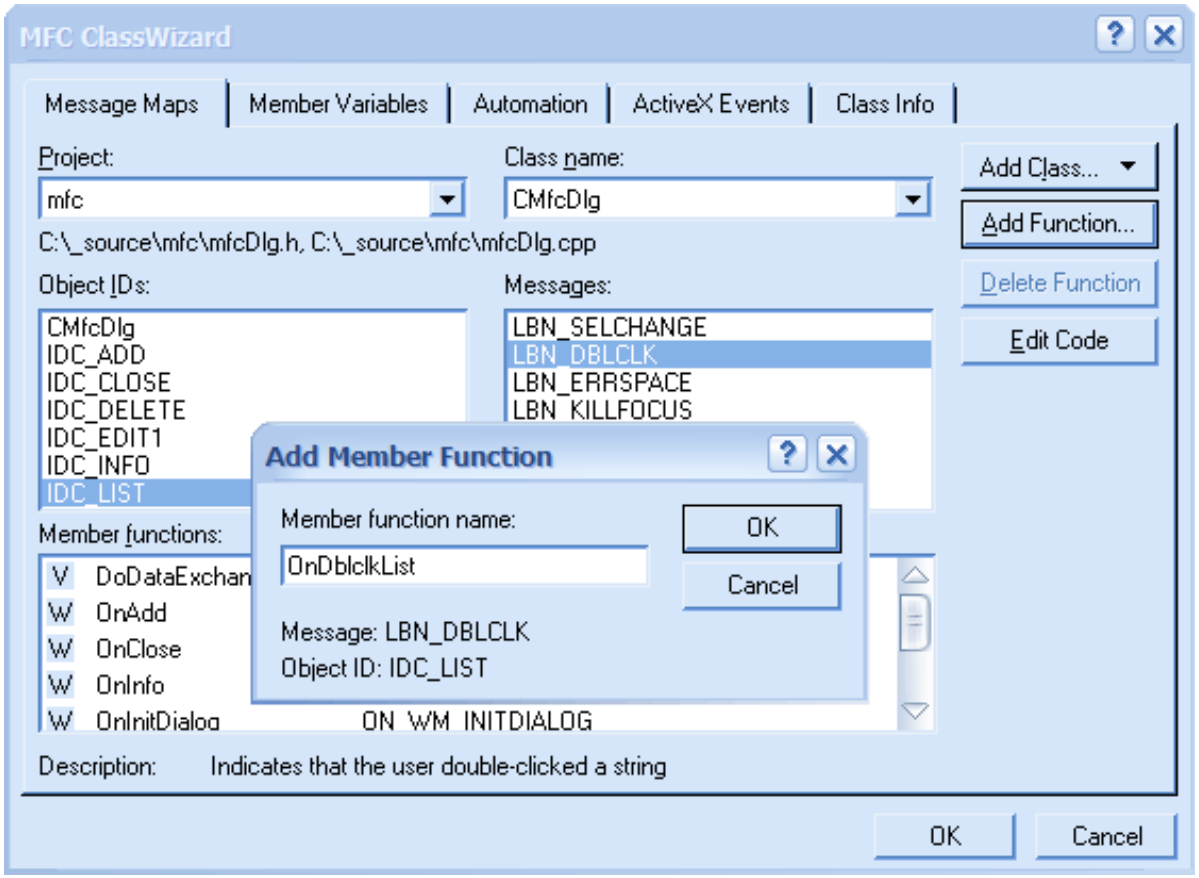
Код обработчика:

```
void CMfcDlg::OnDelete()
{
    if (m_List.GetCurSel() != LB_ERR)
    {
        m_List.DeleteString(m_List.GetCurSel());
    }

    UpdateData();
}
```

`if (m_List.GetCurSel() != LB_ERR)` — проверяем наличие в списке выделенной переменной, для чего вызываем функцию `CListBox::GetCurSel` и если возвращаемая величина не содержит ошибки удаляем выделенную строку с помощью конструкции:

`m_List.DeleteString(m_List.GetCurSel())`, где в качестве параметра функции `DeleteString` выступает индекс выделенной строки.

Добавим обработчик события двойного нажатия левой кнопкой мыши по элементу списка *List Box*. Для этого на вкладке *Message Maps* диалогового окна *MFC Class Wizard* в окне списка *Object IDs* нужно выделить идентификатор списка (*IDC_LIST*) и в списке *Messages* выделить сообщение *LBN_DBLCLK*. Затем нажать кнопку *Add Function*. В появившемся окне ввести имя функции (можно оставить без изменений) и нажать OK.

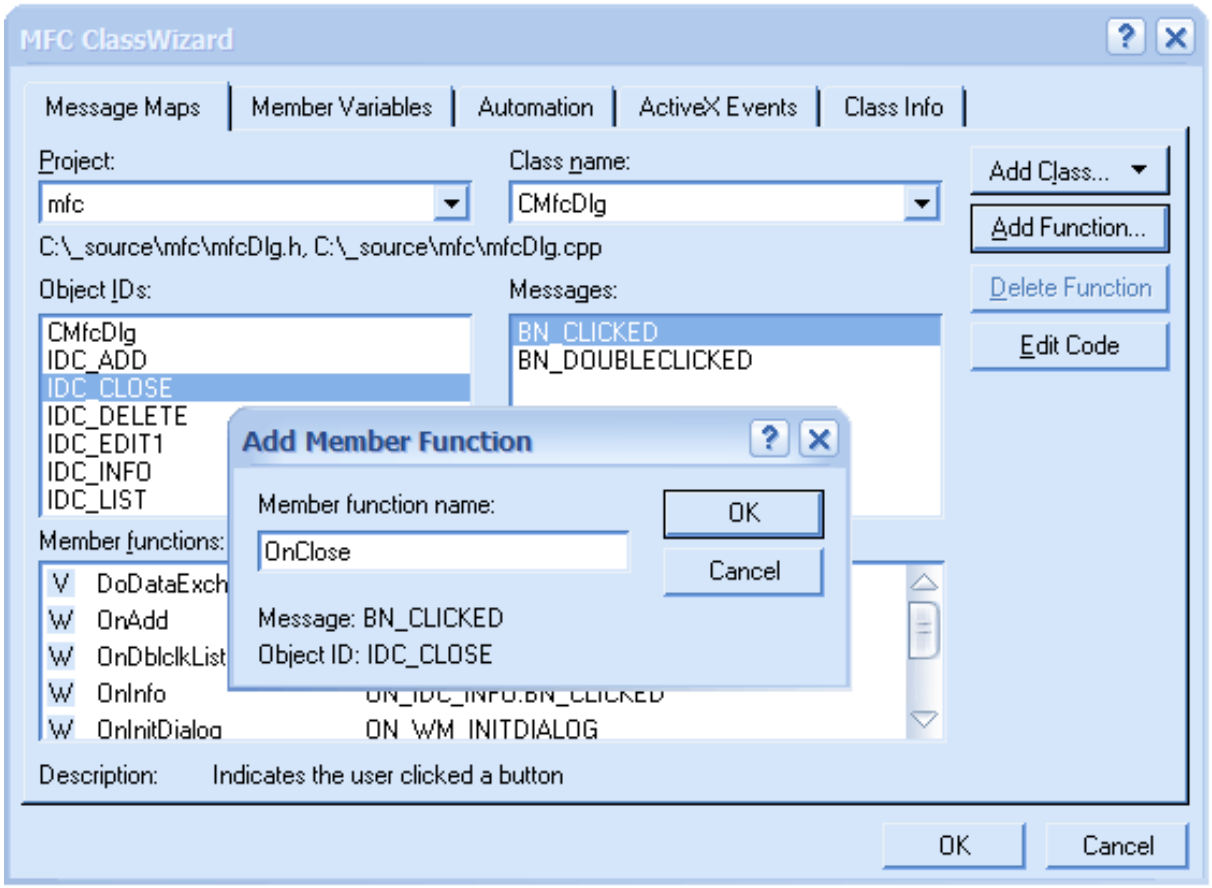


Код обработчика:

```
void CMfcDlg::OnDblclkList()
{
	CString sMystr;

	if (m_List.GetCurSel() != LB_ERR)
	{
		m_List.GetText(m_List.GetCurSel(),sMystr);
		SetDlgItemText(IDC_EDIT1,sMystr);
	}

}
```

Функция **CListBox::OnDblclkList** создает объект класса *CString* с именем **sMystr** и при помощи функции **CListBox::GetText** записывает в него содержимое выделенной строки списка. После чего функция **SetDlgItemText** записывает содержимое переменной **sMystr** в текстовое поле элемента *Edit Box* с идентификатором *IDC_EDIT1*. Если двойной щелчок был произведен на пустой строке списка, и ни один элемент при этом не был выделен, то чтение элемента списка и запись его в текстовое поле не выполняются.

Теперь добавим обработчик события нажатия на кнопку «*Выход*». Для этого на вкладке *Message Maps* диалогового окна *MFC Class Wizard* в окне списка *Object IDs* нужно выделить идентификатор кнопки (*IDC_CLOSE*) и в списке *Messages* выделить сообщение *BN_CLICKED*. Затем нажать кнопку *Add Function*. В появившемся окне ввести имя функции (можно оставить без изменений) и нажать OK.

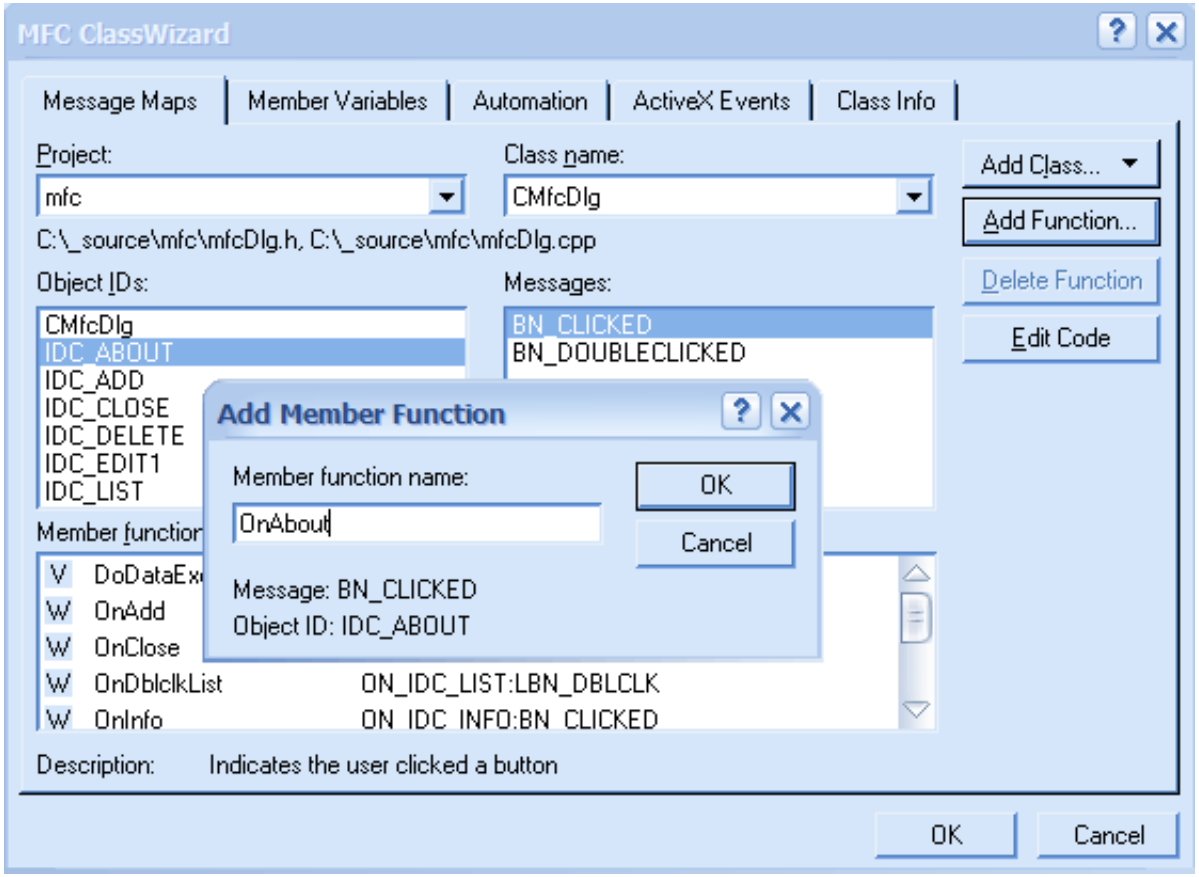


Код обработчика:

```
void CMfcDlg::OnClose()
{
    exit(0);
}
```

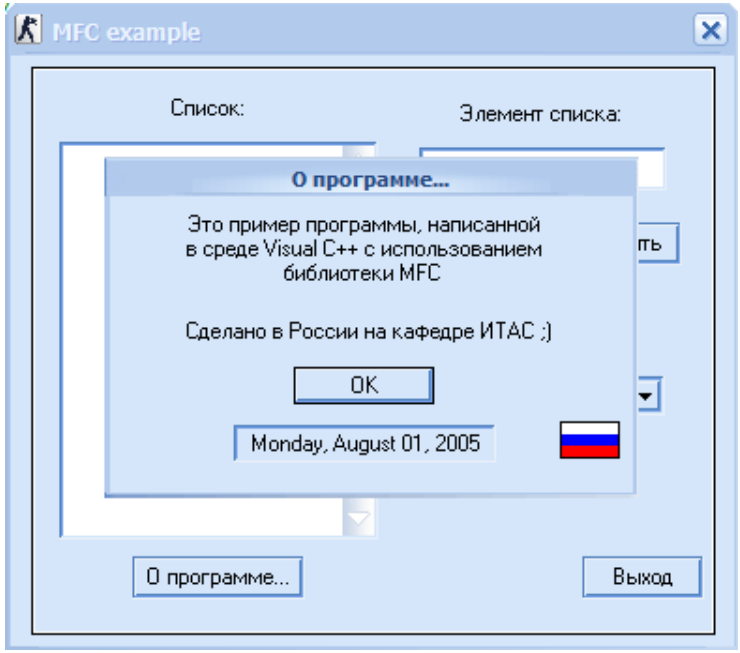
Функция `exit()` завершает текущий процесс, то есть закрывает приложение. Закрыть приложение также можно с помощью конструкции `this -> EndDialog(0)` или функции `ExitProcess(0)`.

Теперь добавим обработчик события нажатия на кнопку «*О программе...*». Для этого на вкладке *Message Maps* диалогового окна *MFC Class Wizard* в окне списка *Object IDs* нужно выделить идентификатор кнопки (*IDC_ABOUT*) и в списке *Messages* выделить сообщение *BN_CLICKED*. Затем нажать кнопку *Add Function*. В появившемся окне ввести имя функции (можно оставить без изменений) и нажать OK.



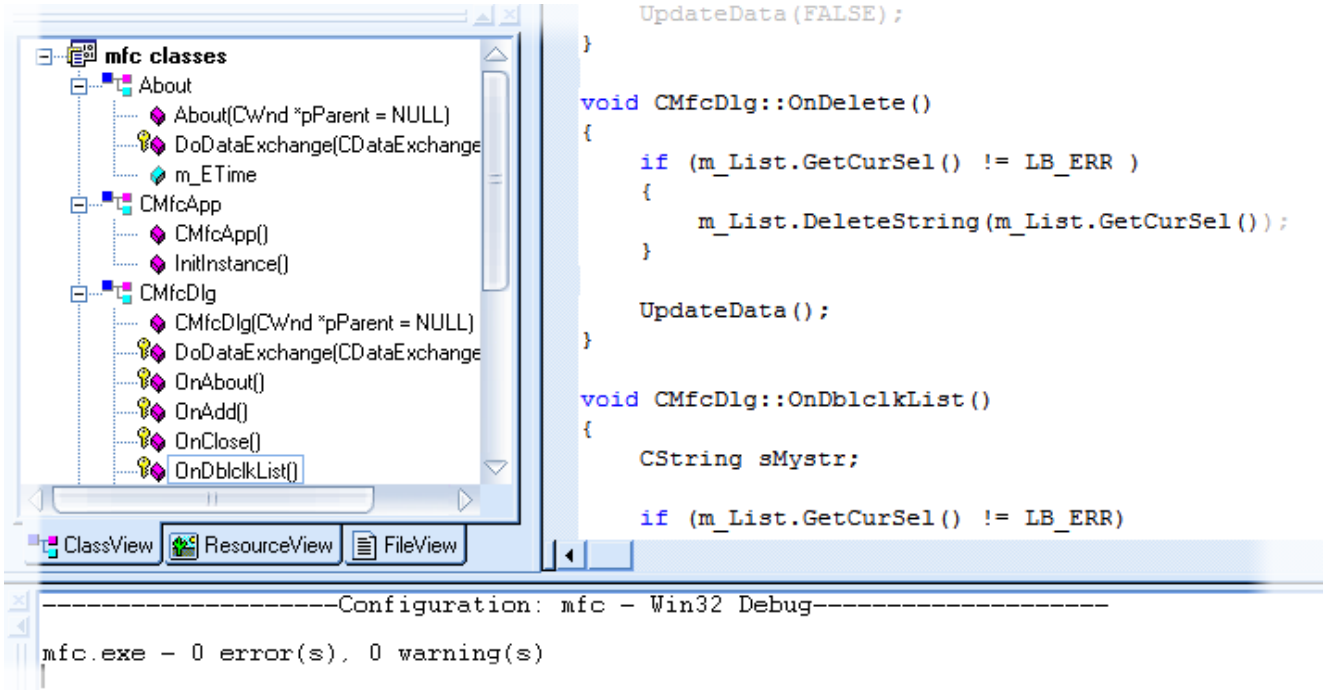
Код обработчика:

```
void CMfcDlg::OnAbout()
{
    About m_About;
    m_About.DoModal();
}
```

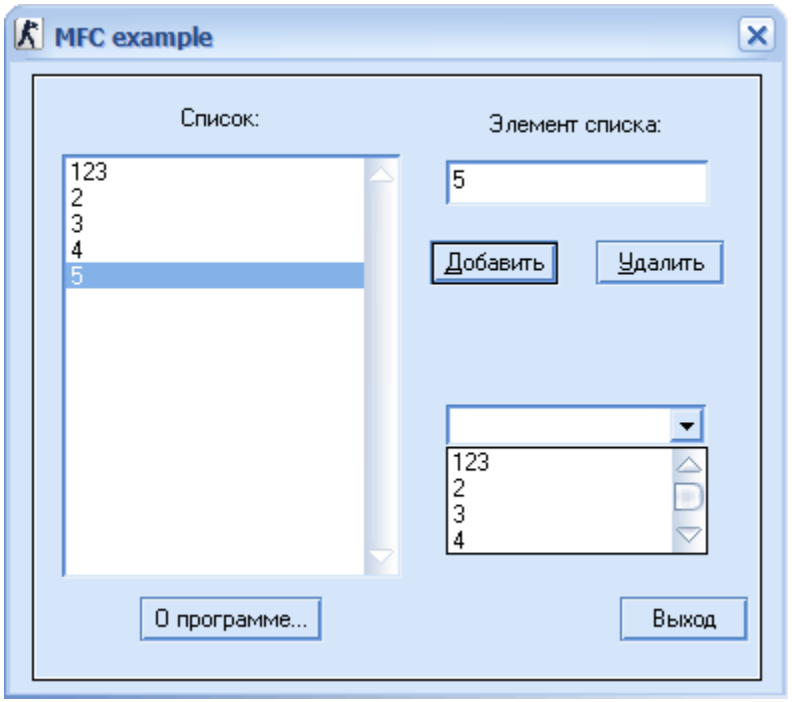


Чтобы вывести модальное диалоговое окно на экран, нужно сначала создать экземпляр класса окна **`m_About`** (класс **`About`** является производным от класса *CDialog*), а затем для экземпляра класса вызвать функцию **`DoModal`**.

Теперь, когда написаны все обработчики событий, можно компилировать проект и запускать приложение:



Приложение компилируется и запускается. Теперь можно работать с программой.



Можно сменить конфигурацию проекта с *Debug* (по умолчанию) на *Release*. При этом размер выходного *.exe* файла изменится с 117 Кб на 28 Кб.

Вопросы к главе

1. Что представляет собой библиотека *MFC* и в чем ее основные преимущества ?
2. Из каких двух функций состоит минимальная программа для *ОС Windows* ?
3. Что такое *интерфейс прикладного программирования* ?
4. Что описывает структура *WNDCLASS* ?
5. Назовите назначение функций *ShowWindow*, *UpdateWindow* ?
6. С помощью какой функции можно вывести сообщение на экран ?
7. Назовите назначение функции *DefWindowProc* ?
8. Какие основные типы приложений позволяет создавать мастер *MFC AppWizard* ?
9. Перечислите основные элементы (компоненты) управления в среде *MS VC++*, назовите их назначение.
10. Как поместить элемент управления на заготовке окна ?
11. Что означает символ *&* перед буквой при установке свойства *Caption* для элемента *Button* ?

12. Как добавить новый ресурс в проект ?

13. Можно ли не компилируя и не запуская проект просмотреть, как в действительности будет выглядеть окно программы ? Если можно, то как это сделать ?

14. Зачем нужны переменные для элементов управления ?

15. Чем между собой отличаются переменные категории *Value* и *Control* для элементов управления ?

16. Как добавить обработчик события нажатия на клавишу ?

17. Для чего нужна функция *UpdateData* ?

18. Как установить фокус ввода на элемент управления ?

19. Какой класс обеспечивает работу со временем в среде *MS VC++* ?

20. Как вывести на экран модальное окно ?