ЛАБОРАТОРНАЯ РАБОТА №1

КАРКАСНОЕ ПРИЛОЖЕНИЕ WINDOWS

ОГЛАВЛЕНИЕ

1. ОБЩАЯ СТРУКТУРА ПРИЛОЖЕНИЙ WINDOWS _____2
   1.1. Функция WinMain() _____2
   2.1. Оконная функция _____7
2. КАРКАСНОЕ ПРИЛОЖЕНИЕ WINDOWS _____10
3. КОМПИЛЯЦИЯ И СБОРКА ПРОГРАММ WIN32 _____17
   3.1. Компиляция приложений Win32 в среде Microsoft Visual Studio____18

1. ОБЩАЯ СТРУКТУРА ПРИЛОЖЕНИЙ WINDOWS

Подавляющее большинство приложений, ориентированных на работу в системах Win32, должно использовать нормальный интерфейс и некоторые общие приемы разработки программ. Стандарты разработки приложений Windows являются следствием архитектурных особенностей систем семейства Windows. Традиционное приложение Windows содержит, как правило, два основных элемента: функцию WinMain() и специальную оконную функцию, предназначенную для взаимодействия с Windows (здесь и далее описание функций, типов данных и примеры программ приводятся в расчете на использование языка C/C++).

1.1. Функция WinMain()

Функция WinMain() является точкой входа в любую программу Windows. Вообще говоря, значение имеет не столько имя функции, сколько ее параметры и тип возвращаемого значения, однако все существующие компиляторы языка C++, предназначенные для разработки программ для Windows, ожидают встретить в тексте программы функцию WinMain(), подобно тому, как компиляторы C/C++ для DOS производили поиск функции main(). Таким образом, для корректной компиляции программы Windows должны как минимум содержать функцию WinMain().

Функция WinMain() определена в стандарте Windows следующим образом:

```
int WINAPI WinMain (HINSTANCE hThisInst,
HINSTANCE hPrevInst,
LPSTR CmdLine,
int ShowMode);
```

Тип возвращаемого значения функции WinMain() определен как WINAPI. Фактически WINAPI представляет собой один из типов, описанных в заголовочном файле `windows.h` посредством директивы #define. Его реальное значение зависит от платформы, для которой разрабатывается приложение, используемого компилятора и ряда других факторов. В приложениях, создаваемых для платформы Intel x86, значением WINAPI является `__stdcall`, что означает функцию, использующую стандартный тип вызова (параметры функции кладутся в стек справа налево, для идентификаторов не генерируются знаки подчеркивания и т.п.). Рекомендуется использовать именно WINAPI как тип возвращаемого значения для WinMain(), в противном случае возможны проблемы при компиляции исходных текстов программ, так как реальное значение WINAPI может измениться при переходе к новым версиям Windows. Функция WinMain() принимает 4 параметра. Первые два определены как HINSTANCE – тип, предназначенный для описания т.н. *дескрипторов приложения*. При этом первый параметр **hThisInst** является дескриптором текущего экземпляра приложения, а **hPrevInst** – дескриптором предыдущего экземпляра приложения. Данные дескрипторы могут быть использованы при вызове тех функций Win32 API, для работы которых требуется информация о том, какое приложение их вызывает. Следует отметить, что параметр **hPrevInst** существует только для совместимости с приложениями Win16 и поэтому всегда равен NULL в программах Win32, а реальное значение имеет только параметр **hThisInst**. Параметр **CmdLine** содержит адрес буфера, содержащего командную строку при вызове приложения. С его помощью можно, например, узнать параметры командной строки, переданные в программу. И, наконец, последний параметр – **ShowMode** определяет режим показа окна программы (свернутое окно, развернутое на весь экран и т.п.). Названия всех перечисленных параметров являются, вообще говоря, произвольными и могут определяться по усмотрению программиста. Важен лишь их тип и порядок следования. Если приложение не использует в своей

работе дескриптор приложения, параметры командной строки и режим показа окна, то допускается, в соответствии с нотацией языка C++, описание функции WinMain() вообще без наименования аргументов, например:

```
int WINAPI WinMain (HINSTANCE, HINSTANCE, LPSTR, int);
```

В простейшем случае функция WinMain() является единственной функцией в приложении. В следующем примере приводится листинг микроскопического приложения Windows, вся работа которого сводится к вызову стандартной функции MessageBox() для вывода соответствующего сообщения в специальном окне для вывода сообщений:

```
//-------------- Простейшее приложение Windows --------------

#include <windows.h>

int WINAPI WinMain (HINSTANCE, HINSTANCE, LPSTR, int)
{
return MessageBox (NULL, "Hello, Windows!",
   "First Windows Program",
   MB_OK | MB_ICONEXCLAMATION);
}
```

Как видно из листинга, функция WinMain() просто возвращает результат выполнения функции MessageBox(). MessageBox() является одной из стандартных функций, входящих в состав Win32 API, поэтому сведения о ее параметрах, можно найти в соответствующей документации (в частности, в файлах справки по C++ Builder/Delphi, находящихся в папке Help/MS SDK Help Files).

Вообще говоря, функция WinMain() может содержать любой легальный с точки зрения операционной системы код, например, вызовы каких–либо

функций Win32 API. Иначе говоря, структура приложения Win32 определяется программистом, и может быть вполне произвольной. Так, к примеру, приложение может содержать функции, выполняющие некоторые действия, а вызовы этих функций могут производиться в теле функции WinMain(), либо WinMain() может включать в себя полностью весь код приложения. ***Каких-либо специальных ограничений на код, находящийся в теле функции WinMain(), при программировании в Win32 не существует!***. Единственным требованием к приложению Windows является обязательное наличие WinMain(). Примером программ подобного рода являются консольные приложения Win32, т.е. полностью 32-разрядные приложения, которым доступны все возможности Win32 API, но которые для операций ввода/вывода используют не окна Windows, а текстовое окно (консоль).

Однако подавляющее большинство приложений Win32 имеет дело именно с графическими окнами Windows. Все приложения, предназначенные для работы с окнами, должны производить некоторый набор базовых операций для создания окна и последующих манипуляций с ним. В этом случае функция WinMain() не просто служит точкой входа в программу, но и должна выполнять определенную работу по инициализации окна. Данное обстоятельство все же накладывает определенные требования на содержание функции WinMain().

Как правило, приложения Win32, ориентированные на работу с окнами, строятся по шаблону, согласно которому функция WinMain() обязательно содержит код, последовательно выполняющий следующие операции:

1. Определение класса окна.
2. Регистрацию класса окна.
3. Создания окна данного класса.
4. Отображение созданного окна.
5. Запуск цикла обработки сообщений.

Рассмотрим эти операции более подробно.

Определение класса окна производится путем заполнения полей специальной информационной структуры типа WNDCLASS (информация о ней содержится в документации Win32), которая описывает некоторые общие характеристики окна, главными из которых является адрес оконной функции, связанной с этим окном, дескриптор приложения, которому принадлежит окно, используемые курсор и иконка, а также имя оконного меню, если таковое имеется.

Регистрация класса окна производится вызовом стандартной функции RegisterClass(), параметром которой является адрес ранее заполненной структуры типа WNDCLASS. В случае успешного завершения класс окна регистрируется в системе, что дает возможность создавать окна данного класса.

Создание окна осуществляется вызовом стандартной функции CreateWindow(), параметрами которой являются имя класса создаваемого окна, экранные координаты окна, стиль окна и т.д. (полный список параметров приводится в документации Win32). Функция CreateWindow() возвращает значение типа HWND – т.н. *дескриптор окна*. Дескриптор окна представляет собой некоторую величину, которая однозначно определяет данное окно. Дескриптор окна применяется при вызове функций, которым требуется информация о том, с каким именно окном они работают (ShowWindow(), SendMessage() и другие). Между созданием окна и определением класса окна существует принципиальное отличие. При определении и регистрации класса окна создается некий обобщенный шаблон, а при создании окна – конкретный экземпляр окна по этому шаблону, подобно тому, как происходит создание объектов класса в ООП. Как следствие, можно, например, создать один класс окна и несколько окон на его основе. Такие окна будут отличаться своими координатами и размерами, но иметь одинаковую форму курсора и использовать общую оконную функцию.

Отображение окна выполняется функцией ShowWindow(), первым параметром которой передается дескриптор ранее созданного окна. Операция создания окна подразумевает только создание в памяти специальной структуры, описывающей окно. По этой причине для вывода окна на экран следует применять функцию ShowWindow(), которая непосредственно рисует окно заданного размера в заданных координатах и заданном режиме.

Запуск цикла обработки сообщений является крайне важным фрагментом WinMain(). Запуск цикла обработки сообщений (messaging loop) завершает инициализацию окна и показывает, что созданное окно готово взаимодействовать с операционной системой. Такой цикл обязательно должен присутствовать во всех оконных приложениях для Windows. Его целью является получение и трансляция сообщений, передаваемых операционной системой. Сообщения ставятся в очередь сообщений приложения, откуда извлекаются программой по мере готовности (как правило, вызовом функции GetMessage()). После получения сообщения оно возвращается программой обратно операционной системе, которая в дальнейшем передает его оконной функции приложения для обработки и выполнения необходимых действий. Цикл обработки сообщений обычно проектируется таким образом, что он завершается при получении сообщения о завершении программы. Таким образом, в подавляющем большинстве приложений Windows завершение цикла обработки сообщений равносильно завершению программы, поэтому после цикла обработки сообщений в функции WinMain() как правило стоит единственный оператор возврата return, завершающий выполнение WinMain().

2.1. Оконная функция

Оконная функция является специальной функцией, которую в обязательном порядке должны иметь все приложения, работающие с окнами.

Особенности архитектуры оконной подсистемы в Windows таковы, что оконное Win32-приложение, не имеющее оконной функции, просто не сможет взаимодействовать с операционной системой, а, значит, потеряет всякий смысл с точки зрения идеологии Windows.

Оконная функция относится к типу т.н. ***функций обратного вызова (callback)***. Дело в том, что оконная функция вызывается не приложением, а операционной системой. Используя эту функцию, Windows взаимодействует с программой. Схема взаимодействия операционной системы с приложением при вызове оконной функции выглядит следующим образом: Windows ставит некоторое адресованное программе сообщение в очередь сообщений программы, программа в цикле обработки сообщений забирает это сообщение из очереди и снова отправляет его операционной системе, сигнализируя, что сообщение получено и транслировано, тогда операционная система вызывает оконную функцию, передавая ей сообщение в качестве параметра, а оконная функция выполняет обработку сообщения. Одна из причин использования этой достаточно сложной схемы заключается в поддержке оконной подсистемой вытесняющей многозадачности, при которой работающее приложение в любой момент может быть прервано планировщиком задач. Оконная функция вызывается операционной системой каждый раз, когда Windows передает сообщение окну приложения.

Предназначение оконной функции заключается в обработке полученных сообщений. Сообщения в оконную функцию передаются через один из параметров. Принимая сообщения, посылаемые операционной системой, оконная функция должна выполнять действия, соответствующие типам и параметрам этих сообщений. Во многих случаях, тело оконной функции представляет собой просто большой оператор `switch`, определяющий тип получаемых сообщений и выполняющий соответствующие действия для каждого типа. Программа не обязана выполнять обработку всех типов сообщений, посылаемых операционной системой, поскольку таких типов существует несколько сотен. Сообщения,

которые не обрабатываются программой, должны обрабатываться Windows по умолчанию. Большая часть сообщений именно так и обрабатываются. Для этого они пересылаются в оконную подсистему посредством вызова специальной функции DefWindowProc().

Оконная функция может иметь произвольное название, но должна иметь строго определенное количество и типы параметров, а также тип возвращаемого значения. Оконная функция обычно определяется следующим образом:

```
LRESULT CALLBACK WinProc (HWND wnd,  UINT msg,
                          WPARAM wp, LPARAM lp);
```

Параметры оконной функции определяются следующим образом:

- HWND wnd – дескриптор окна, которому принадлежит оконная функция.
- UINT msg – идентификатор сообщения (32-разрядное число – номер сообщения).
- WPARAM wp, LPARAM lp – параметры сообщения, их смысл зависит от типа сообщения.

Оконная функция имеет тип возвращаемого значения LRESULT CALLBACK. При этом LRESULT определен в Windows как __int32 (для 32-разрядных версий Windows), но, в принципе, результатом выполнения оконной функции чаще всего становится логическое значение. Общепринято поэтому, что оконная функция возвращает 1 в случае успешной обработки какого-либо сообщения и 0 – в случае неудачи, хотя может возвращать и любое другое значение по усмотрению программиста. Тип CALLBACK определен в Windows просто как __stdcall, но его определение может измениться в будущих версиях Windows. Рекомендуется оконную функцию

всегда определять именно как LRESULT CALLBACK для предотвращения возможных конфликтов в будущем.

2. КАРКАСНОЕ ПРИЛОЖЕНИЕ WINDOWS

Как уже отмечалось ранее, большинство Win32-программ имеют определенные общие черты. На примере следующей программы демонстрируются основные принципы работы оконных приложений Win32. Данная программа может служить шаблоном или ***каркасом***, т.е. после соответствующей доработки может применяться для решения других задач.

```
//--------------- Каркасное приложение Windows ---------------
     #include <windows.h>

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

     WINAPI WinMain (HINSTANCE hInst, HINSTANCE, LPSTR, int)
     {
        WNDCLASS wcl;
        MSG      msg;
        HWND     wnd;

        wcl.style  = 0;
        wcl.lpfnWndProc = WndProc;
        wcl.hInstance  = hInst;
        wcl.hCursor = LoadCursor( NULL, IDC_ARROW);
        wcl.hIcon = LoadIcon (NULL, IDI_APPLICATION);
        wcl.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
        wcl.lpszMenuName = NULL;
        wcl.lpszClassName = "First";
        wcl.cbClsExtra = wcl.cbWndExtra = 0;

        if (!RegisterClass (&wcl)) return 1;

        wnd  = CreateWindow
```

```
   (
     "First",
     "Hello, Windows!",
      WS_OVERLAPPEDWINDOW,
      CW_USEDEFAULT, CW_USEDEFAULT,
      CW_USEDEFAULT, CW_USEDEFAULT,
      NULL, NULL,
      hInst, NULL);

   ShowWindow (wnd, SW_SHOW);

   while (GetMessage (&msg, NULL, 0, 0))
   {
       TranslateMessage (&msg);
       DispatchMessage (&msg);
   }

   return msg.wParam;
}

LRESULT CALLBACK WndProc (HWND wnd, UINT msg,
                          WPARAM wp, LPARAM lp)
{
   switch( msg )
   {

      case WM_DESTROY:
         PostQuitMessage(0);
         return 0;

      default:
         return DefWindowProc( wnd, msg, wp, lp );
   }
}
```

Теперь рассмотрим более подробно работу представленного каркасного приложения и его отдельных фрагментов.

Прежде всего, все программы для Win32 должны включать в себя заголовочный файл **windows.h**, который содержит все необходимые определения типов данных Windows и описание прототипов функций API. В данном примере это делается в первой строке программы с помощью директивы **`#include <windows.h>`**.

Далее в каркасном приложении следует прототип оконной функции. Согласно правилам языка C++, прототип функции должен быть описан в тех случаях, когда необходимо обращаться к функции до ее определения. В данном случае имя оконной функции указывается при регистрации класса окна в WinMain(), а само тело функции описано далее в программе, поэтому для предотвращения ошибки компиляции необходимо указать прототип оконной функции. В приведенном примере оконная функция имеет имя **WndProc** и стандартный набор параметров, а также возвращаемое значение LRESULT CALLBACK.

После прототипа оконной функции в программе идет функция WinMain(), описанная в соответствии со стандартом. Из всех параметров WinMain() используется только дескриптор текущего приложения, поэтому только он имеет собственное имя **hInst**, для остальных же параметров указаны лишь их типы. Тело функции WinMain() начинается с описания 3 переменных: переменной **WNDCLASS wcl** – структуры для описания класса создаваемого окна, переменной **MSG msg** – структуры, которая используется для хранения передаваемых в программу сообщений, а также переменной **HWND wnd** – дескриптора окна программы. Далее в функции WinMain() выполняется операция создания класса окна. Создание класса окна заключается в заполнении полей структуры WNDCLASS:

```
wcl.style  = 0;
wcl.lpfnWndProc = WndProc;
wcl.hInstance  = hInst;
wcl.hCursor = LoadCursor( NULL, IDC_ARROW);
wcl.hIcon = LoadIcon (NULL, IDI_APPLICATION);
wcl.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
wcl.lpszMenuName = NULL;
wcl.lpszClassName = "First";
wcl.cbClsExtra = wcl.cbWndExtra = 0;
```

Как видно, полю **style** присваивается значение 0, что означает стандартный стиль окна (другие стили описаны в документации Win32), полю **lpfnWndProc** – адрес оконной функции приложения (в данном случае – WndProc), а полю **hInstance** – значение дескриптора приложения, переданного в качестве параметра в WinMain(), т.е. **hInst**. Поля **lpszMenuName** и **lpszClassName** соответственно хранят в себе имя меню программы, если оно присутствует (в данном случае меню нет, поэтому полю присваивается значение NULL), и строку с именем класса окна. Имя класса однозначно идентифицирует класс окна в системе, в приведенном листинге класс получает имя “First”. Поля **cbClsExtra** и **cbWndExtra** предназначены для хранения некоторых дополнительных параметров окна и обычно равны 0. Поля **hCursor, hIcon** и **hbrBackground** определяют визуальные параметры окна: тип курсора, тип иконки и цвет фона окна соответственно. Все эти поля являются дескрипторами соответствующих элементов: курсора, иконки и кисти. Для заполнения данных полей приложение может использовать дескрипторы собственных ресурсов, либо встроенных ресурсов Windows. В каркасном приложении используются стандартные ресурсы Windows. В частности, курсор и иконка определяются вызовом однотипных функций LoadCursor() и LoadIcon(), которые принимают NULL в качестве первого параметра (что означает использование системных ресурсов) и имя ресурса, в котором описаны курсор и иконка, а возвращают дескрипторы этих

элементов. Стандартные курсоры и иконки Windows имеют собственные символьные имена, которые описаны в **windows.h**. Каркасное приложение использует стандартный курсор в виде стрелки (**IDC_ARROW**) и стандартную иконку с логотипом Windows (**IDI_APPLICATION**). Значения дескрипторов, полученные после вызовов LoadCursor() и LoadIcon(), помещаются в поля hCursor и hIcon соответственно. Для заполнения поля дескриптора кисти **hbrBackground**, который используется для определения цвета фона окна, применяется стандартная системная константа **COLOR_WINDOW**. Полученное с ее помощью значение явно приводится к типу дескриптора кисти HBRUSH и присваивается полю hbrBackground.

Когда класс окна полностью определен, он должен быть зарегистрирован в системе. Регистрация выполняется с помощью функции API RegisterClass(), которая принимает в качестве параметра адрес ранее заполненной структуры типа WNDCLASS и возвращает некоторое значение, однозначно идентифицирующее созданный класс в системе. Возврат 0 при вызове RegisterClass() показывает, что произошла ошибка при регистрации класса окна. В этом случае продолжение работы не имеет смысла и приложение, перехватив эту ошибку, немедленно завершается.

В случае успешного создания и регистрации класса окна, приложение может создавать окна данного класса. Окна создаются, как правило, вызовом функции CreateWindow(), которая возвращает дескриптор созданного окна (в данном примере он заносится в переменную wnd). Параметрами функции CreateWindow() являются соответственно имя класса окна (в примере – “First”), строка заголовка окна (в примере – "Hello, Windows!"), тип окна (в данном случае – **WS_OVERLAPPEDWINDOW**, что означает стандартное окно, имеющее заголовок, системное меню, кнопки минимизации, развертки и закрытия и т.п.), координаты X, Y верхнего левого угла окна на экране и длина и ширина окна (для всех этих параметров можно применить стандартную константу **CW_USEDEFAULT**, означающую, что все эти параметры будут выбраны автоматически самой системой), дескриптор

родительского окна (равен NULL или **HWND_DESKTOP** если нет окна, породившего данное) и дескриптор меню (также равен NULL, если нет меню), дескриптор приложения, которому принадлежит окно (в данном примере равен hInst) и указатель на некоторую структуру с дополнительной информацией об окне (как правило, равен NULL, поскольку обычно не используется). Как видно, большинство параметров при вызове ShowWindow() могут быть установлены в значения по умолчанию, обязательными являются лишь имя класса окна (1-й параметр), стиль окна (3-й параметр) и дескриптор приложения (10-й параметр). Кроме того, обычно в заголовке окна присутствует какой-либо текст, поэтому при вызове CreateWindow() указывается и текст заголовка окна (2-й параметр).

Как уже отмечалось ранее, создание окна еще не означает его немедленного отображения на экране. Для этой цели используется функция ShowWindow(). Ее первым параметром является дескриптор отображаемого окна (в каркасном приложении используется переменная wnd, значение которой было установлено предыдущим вызовом CreateWindow()). Вторым параметром при вызове ShowWindow() указывается режим отображения окна, который может принимать одно из следующих значений:

- **SW_HIDE –** окно не отображается вообще
- **SW_MINIMIZE –** окно сворачивается в Панель задач (Taskbar)
- **SW_MAXIMIZE –** окно разворачивается на полный экран
- **SW_SHOW –** окно отображается в обычном виде

В каркасном приложении применяется последняя константа, заставляющая окно отображаться в нормальном виде.

Последней частью функции WinMain() является цикл обработки сообщений. В каркасном приложении он реализован в виде цикла `while`, в котором вызывается функция API GetMessage(). Эта функция проверяет очередь сообщений приложения и, если очередь пуста, возвращает

управление Windows. Если же в очереди есть сообщения, то они извлекаются по порядку и помещаются в ранее описанную структуру msg, которая является буфером для хранения информации о сообщении. Адрес структуры msg передается первым параметром в функцию GetMessage(). Вторым параметром GetMessage() принимает дескриптор окна, которому направляется получаемое сообщение. Это позволяет фильтровать принимаемые приложением сообщения. В случае, если необходимо обрабатывать все сообщения, адресованные приложению, второй параметр GetMessage() должен быть равен NULL. Следующие два параметра определяют диапазон принимаемых сообщений. Поскольку идентификаторы сообщений являются просто 32-разрядными величинами, то, указав в третьем и четвертом параметрах GetMessage() соответствующие значения, можно ограничить диапазон принимаемых программой сообщений. Для того, чтобы принимать все сообщения (что чаще всего и бывает), эти параметры должны быть равны 0.

Функция GetMessage() устроена так, что возвращает нулевое значение при получении специального сообщения **WM_QUIT** и ненулевое – во всех остальных случаях. Сообщение **WM_QUIT** сигнализирует о завершении работы приложения, поэтому завершение цикла `while` будет фактически означать завершение работы приложения. В этом случае останется только завершить выполнение функции WinMain() (а, значит, и всей программы) оператором `return` с указанием возвращаемого значения.

Внутри цикла обработки сообщений последовательно вызываются функции TranslateMessage() и DispatchMessage(). Функция TranslateMessage() предназначена для трансляции виртуальных кодов клавиш, которые генерируются системой, в клавиатурные сообщения. Использование этой функции не является строго обязательным, однако, она применяется в большинстве случаев, поскольку позволяет использовать в программе ввод с клавиатуры. Функция DispatchMessage() является гораздо более важной. Она возвращает полученное сообщение обратно в Windows и тем самым

инициирует вызов оконной callback-функции. При отсутствии DispatchMessage() в теле WinMain() приложение не будет иметь взаимосвязи с системой и не сможет реагировать ни на какие события, поскольку не будет производиться вызова оконной функции.

После описания WinMain() в каркасном приложении содержится описание оконной функции. Ее основой является оператор `switch`, который анализирует тип переданного сообщения и, в зависимости от типа, выполняет те или иные действия. Вообще говоря, в полноценном приложении такой оператор может быть очень сложным, однако в каркасной программе `switch` очень короткий: он обрабатывает единственное сообщение **WM_DESTROY**, которое посылается операционной системой приложению, когда пользователь закрывает окно (например, щелкнув по кнопке закрытия окна). Данное сообщение должно быть обработано специальным образом, поскольку закрытие окна не подразумевает автоматического завершения программы. Именно поэтому оконная функция, перехватив **WM_DESTROY**, вызывает специальную функцию PostQuitMessage(), которая приводит к появлению в очереди сообщений приложения сообщения **WM_QUIT**, т.е. к завершению программы. Все остальные сообщения, которые посылаются, например, при перерисовке содержимого окна, изменении его размеров, перемещении и т.п. обрабатываются по умолчанию. Для этого оконная функция вызывает стандартную функцию API DefWindowProc(), передавая ей свои параметры.

3. КОМПИЛЯЦИЯ И СБОРКА ПРОГРАММ WIN32

После создания исходного текста приложения Win32 необходимо выполнить компиляцию и компоновку программы для получения исполняемого модуля (файла .EXE или .DLL). Конкретная последовательность действий зависит от типа разрабатываемого приложения (одно- или многопоточное, .EXE или .DLL), вида компоновки (статическая

или динамическая), используемого компилятора (C++ Builder, Visual C++ и т.п.) и используемой технологии программирования (чистый Win32 API, библиотеки классов типа VCL, MFC и т.п.). Следует отметить, что системы быстрой разработки приложений (в частности, Visual C++ и C++ Builder), как правило, сильно автоматизируют процесс компиляции программы, скрывая от программиста отдельные его стадии. Кроме того, разработка приложений с использованием указанных средств, предполагает использование соответствующих библиотек, а, значит, и собственных приемов создания программ. Так, например, нетрудно заметить существенные различия в исходных текстах приложений, написанных на C++ Builder или на Visual C++. Между тем, все современные системы разработки поддерживают стандарт Win32 API и позволяют создавать программы, работающие только с функциями Windows, без использования библиотек классов. Такие программы зачастую проще и удобнее компилировать с помощью компиляторов командной строки, чем из IDE, хотя не исключено и обратное.

3.1. Компиляция приложений Win32 в среде Microsoft Visual Studio

Для создания каркасного приложения Windows в среде разработки Microsoft Visual Studio 2010 необходимо создать соответствующий проект, как показано на приведенных ниже снимках экранов. Созданный проект может быть затем откомпилирован и собран в исполняемый файл средствами Visual Studio.

Для создания проекта необходимо выбрать соответствующий пункт в меню «Файл» (или нажать комбинацию клавиш Ctrl+Shift+N), в появившемся диалоге выбрать тип проекта «Проект Win32» как показано на рис. 1. Кроме того, необходимо также указать название проекта и указать место его расположения в соответствующих полях.

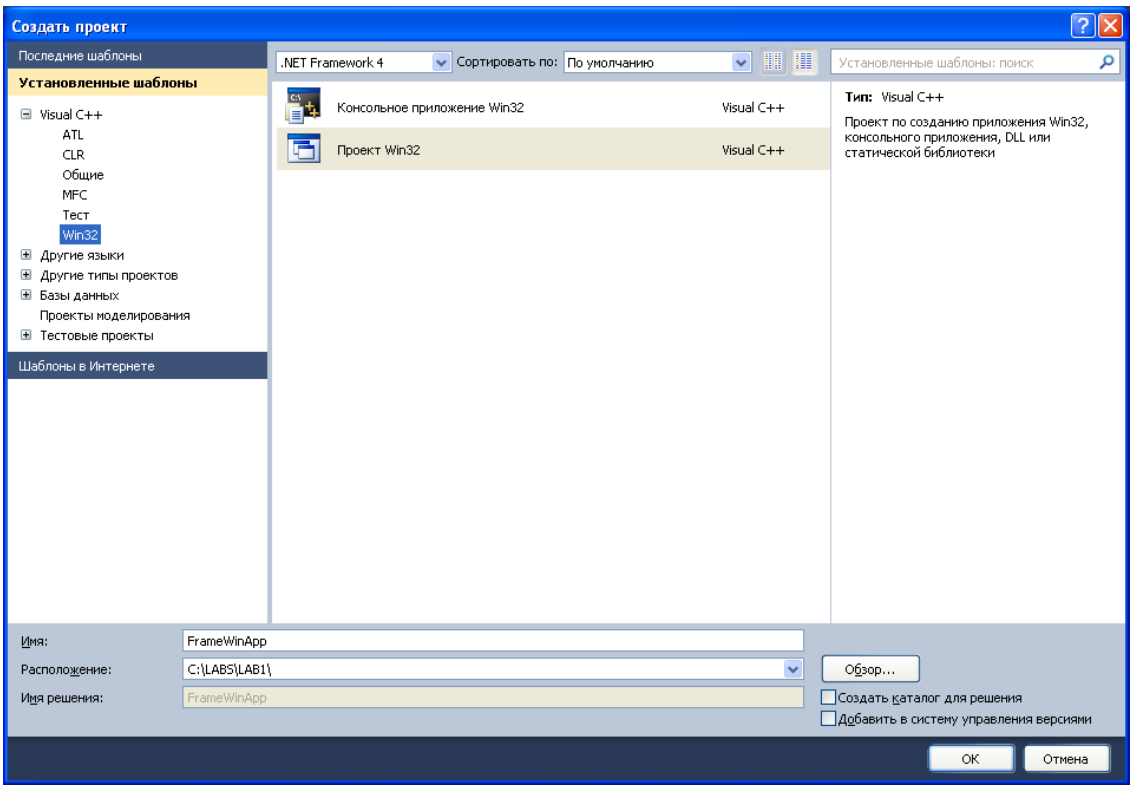

Рис. 1 – Выбор типа проекта

В появившемся окне Мастера приложений Win32 необходимо установить параметры, как показано на рис. 2.

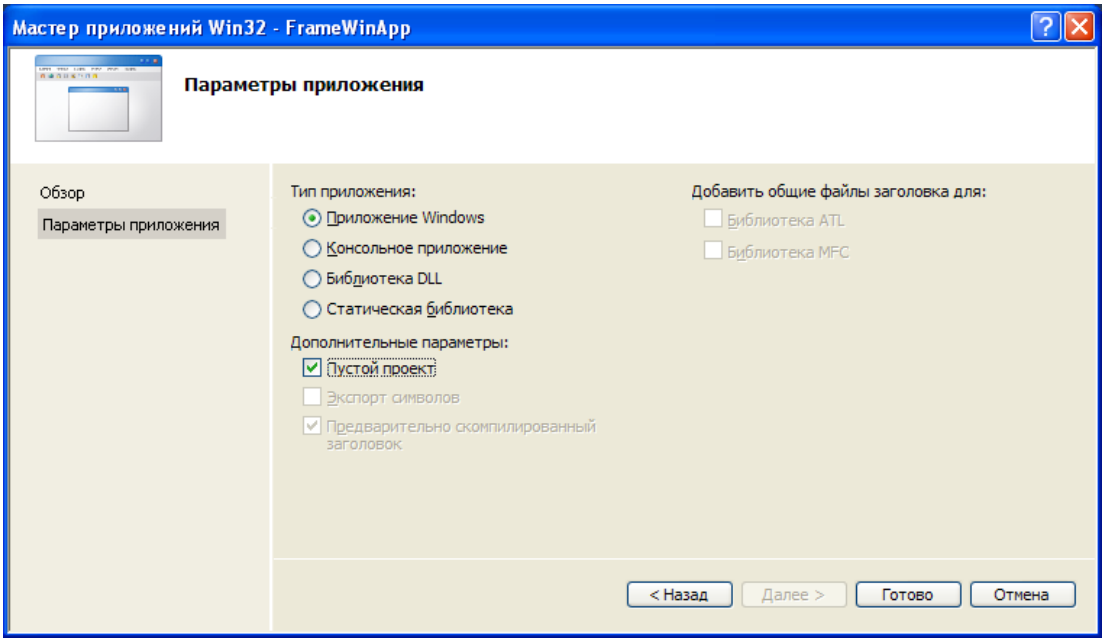

Рис. 2 – Установка параметров проекта.

После выполнения этих действий будет создан пустой проект (рис. 3), в который следует добавить файл, содержащий исходный код каркасного приложения.

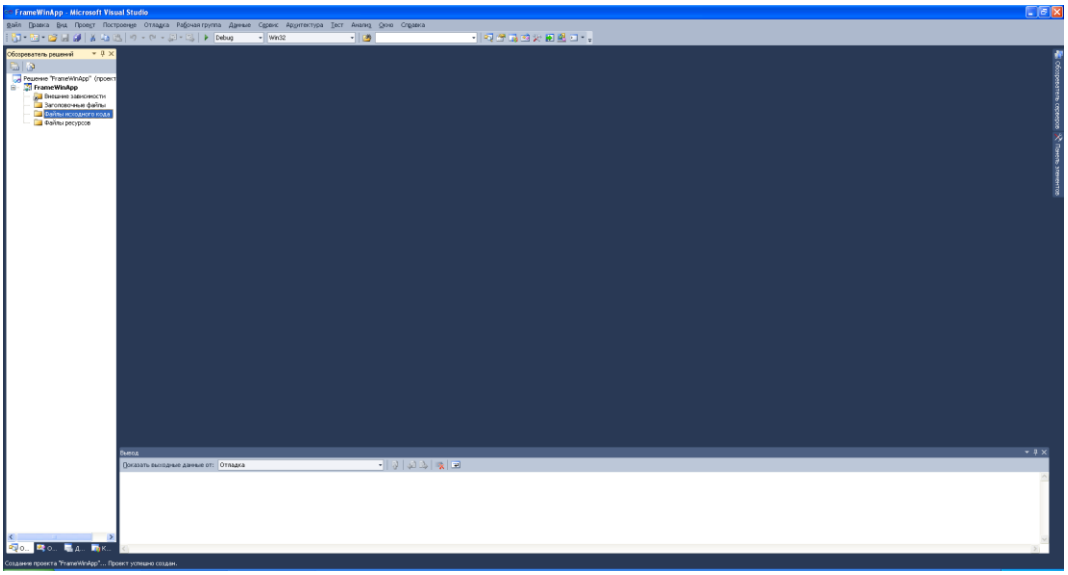

Рис. 3 – Окно созданного пустого проекта

Для добавления файла с исходным кодом следует выбрать соответствующий пункт в меню «Проект» (или нажать комбинацию клавиш Alt+Shift+A), затем в появившемся диалоге выбрать нужный файл.

После добавления файла с исходным кодом каркасного приложения (рис. 4) можно будет выполнить сборку и запуск приложения, нажав клавишу F5 или выбрав соответствующий пункт в меню «Отладка».

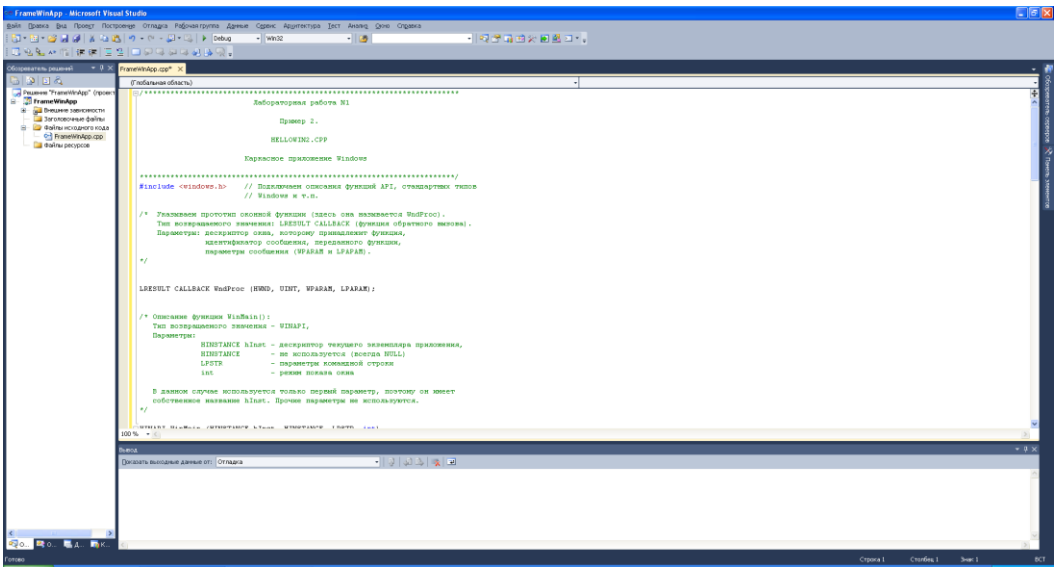

Рис. 4 – Проект готов к сборке и запуску