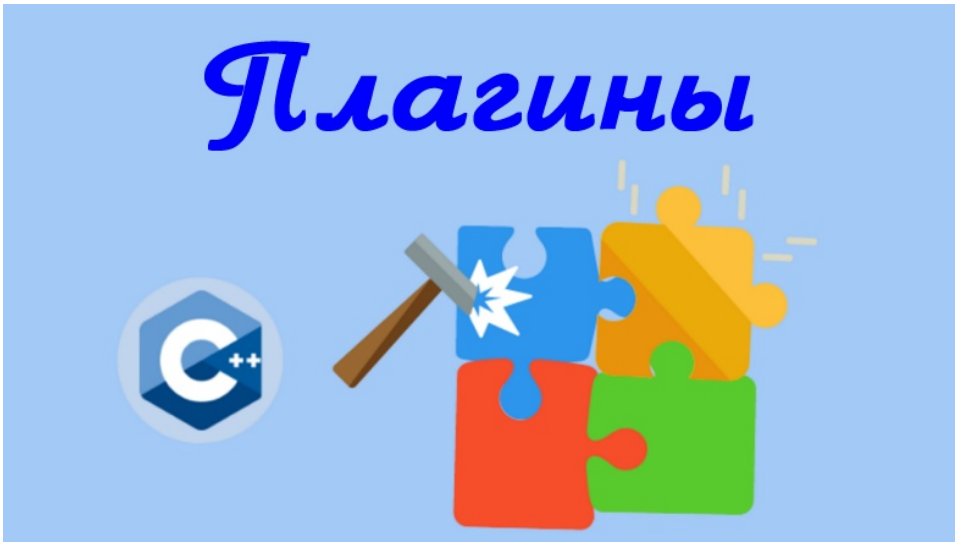
Модульное программирование в C++. Статические и динамические плагины

https://habr.com/ru/post/566864/

Версия pdf-файла: 1.00 от 09.07.2021



На тему модульности программ в C++, в Интернете, теоретических материалов много, а практических – мало. Поэтому, не найдя подходящего прототипа для собственного проекта, пришлось изобретать очередной «велосипед», из чего-то, ведь, исходить надо.

Сложность программирования, тестирования, отладки, поддержки и сопровождения программных продуктов заставляет искать пути борьбы с ней. Одним из традиционных способов является использование плагинов и сервисов с виртуальными интерфейсами. Однако доступные примеры предпочитают иметь дело с консолью, тогда как нам интересны окна и другие компоненты из арсенала GUI (графического интерфейса пользователя).

Часть первая. Результирующая

Введение

Предполагаем, что плагины реализуют оконные компоненты, а сервисы, как правило, безоконные либо вызывают служебные окна и диалоги. Их использование – вполне рабочая идея, но пути возможной реализации могут быть разными.

В нашем случае, удобно, разделить плагины на два вида: статические и динамические. К первым мы отнесем dll раннего связывания, т.е., бинарные модули, известные и доступные на этапе компиляции. А вторые это dll позднего связывания, загружаемые автоматически, из определенного каталога, во время работы основной программы.

Отличие плагинов, особенно динамических, от обычных dll, заключается, как правило, в использовании первыми виртуальных интерфейсов, о которых речь ниже. Единообразие подобных интерфейсов как раз и служит способом их подключения и использования во время работы программного кода.

В принципе, все приложение можно свести к плагинам, оставив только служебный код для подключения модулей. Естественно, что создаваемый либо отлаживаемый модуль должен быть представлен явно своим исходным кодом. При этом, в главном модуле, сам текст программы будет минимальным. Другими словами, для отладки и проверки работы своего модуля в общем приложении, можно использовать главный модуль, в виде простого исходного кода и свой собственный код. Все остальные динамические и статические плагины могут оставаться в своем бинарном виде.

Язык C++ очень удобен для решения задач подобного рода. Но, чтобы не усложнять демонстрационную программу различного рода фреймворками, мы ограничимся использованием только WinAPI.

Таким образом, в данной статье будет рассматриваться программа, ориентированная на классический графический интерфейс пользователя и выполняющая некоторые демонстрационные функции, основанные на программной логике.

Но сначала пара слов о модульности, как таковой.

Модульное структурирование кода

Под модульностью программ мы понимаем, прежде всего, возможность представления части исходного кода проекта, в виде бинарных независимых модулей. Более того, весь существенный код можно вынести во внешние бинарные модули, тогда как оставшийся код будет выполнять, по сути, только служебные функции.

Подобное разделение кода удобно для тестирования, отладки и быстрой компоновки отдельных частей приложения. Это может иметь смысл и для командной работы, когда каждый создает собственный модуль, а потом просто проверяет его работу путем копирования своей библиотеки в соответствующий каталог плагинов.

Однако может возникнуть ситуация, когда необходимо весь исходный код плагинов скомпилировать вместе с основным модулем. Технически это не должно вызывать проблем. Выбор должен быть на уровне параметра условной компиляции. Проще всего, просто сформировать два проекта, один для полной динамической сборки, а второй, для полной статической сборки проекта. Понятно, что допустимы и промежуточные варианты.

Результат работы программы без динамических плагинов

В нашем демо-проекте, мы использовали оба вида компиляции. Естественно, что результат работы у них один и тот же. Только, в динамическом проекте, можно видеть работу программы вообще без плагинов (рис. 1).

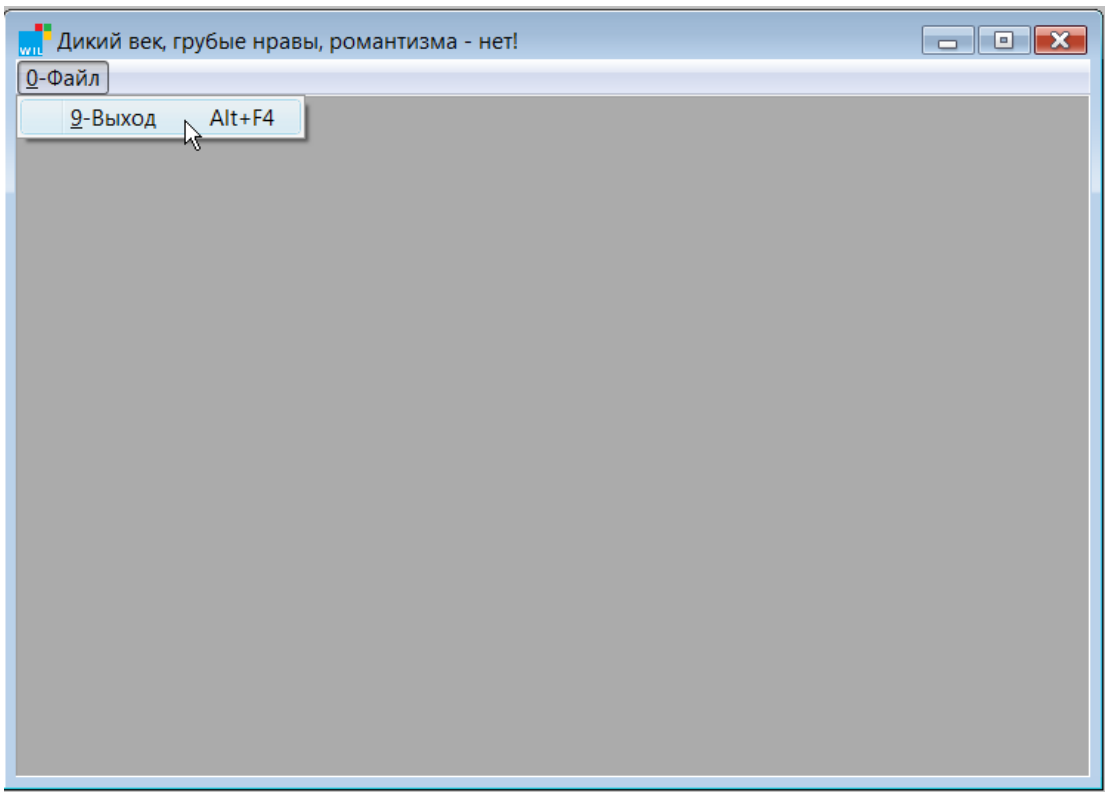


Рис.1. Главное окно приложения при отсутствии плагинов.

В проекте нам доступны два динамических плагина и три статических. Статические плагины присутствуют всегда, либо внутренним образом (при полной статической сборке проекта), либо в виде трех dll (расположенных рядом с exe-модулем):

– Common.dll (библиотека общего назначения, для упрощенной работы с ini-файлами, создания исходных каталогов и т.п.);
– DllLoader.dll (загрузчик динамических плагинов, который оказался достаточно сложным для включения его в библиотеку общего назначения);
– App.dll (основной код, создающий главное окно, организующий работу цикла сообщений и т.п.).

Работа этих статических плагинов (за исключением создания прототипа главного окна) на рисунке не видна, поскольку они предназначены, в основном, для обслуживания динамических плагинов.

Сами же динамические плагины представлены двумя файлами (в папке «Plugins»):

– NewWin.dll (создание множества различных дочерних MDI-окон, одного класса)
и
– About.dll (единственное MDI-окно, эмулирующее диалоговое окно «О программе»).

Заметим, что эти плагины имеют собственные внешние ресурсы в папке «Plugins\Res».

Результат работы программы с динамическими плагинами

Если добавить плагин «NewWin.dll» в папку «Plugins», то увидим следующие изменения (рис.2).

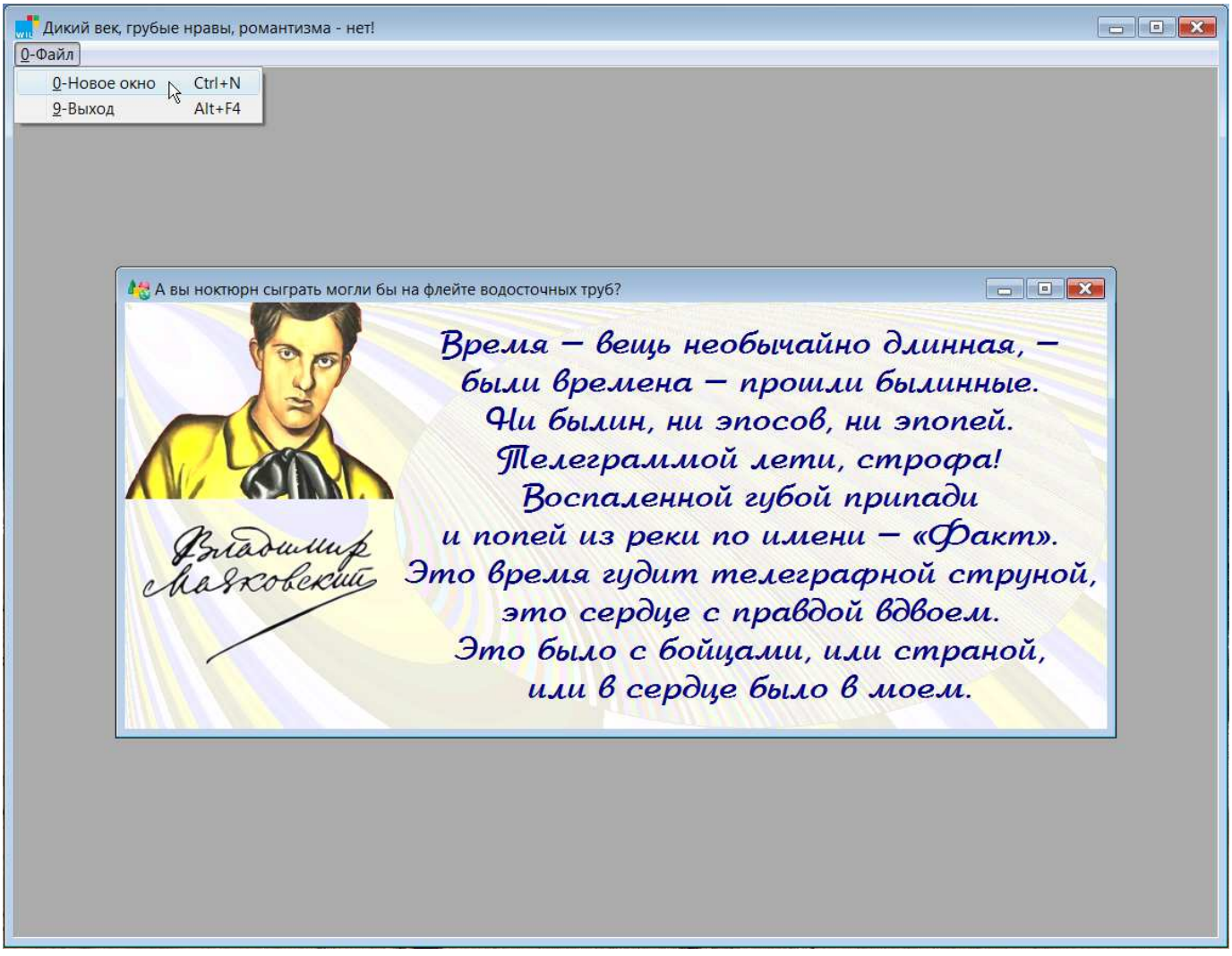


Рис.2. Главное окно приложения при наличии плагина «NewWin.dll».

Этот плагин создает несколько различных окон с цитатами Владимира Маяковского. При этом, первое окно центрируется, а последующие располагаются по диагонали, со смещением (рис. 3).

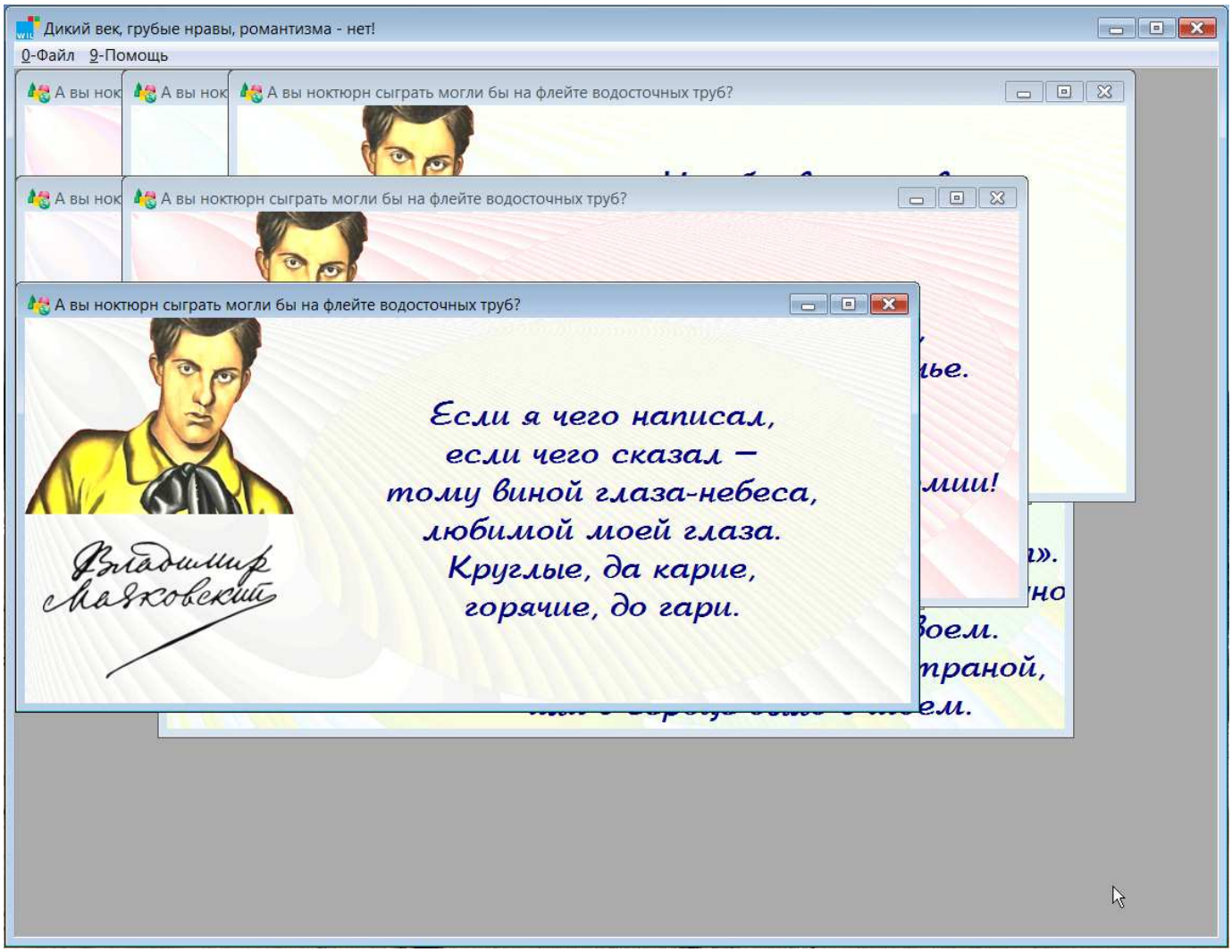


Рис.3. Множество окон плагина «NewWin.dll».

Если изменить размеры текущего окна, то следующее окно тоже изменится соответственно, только на отступы это не повлияет.

Хотя это и не принципиально, но мы ограничили количество одновременно открытых окон, при превышении которых будет выдано предупреждение (рис. 4).

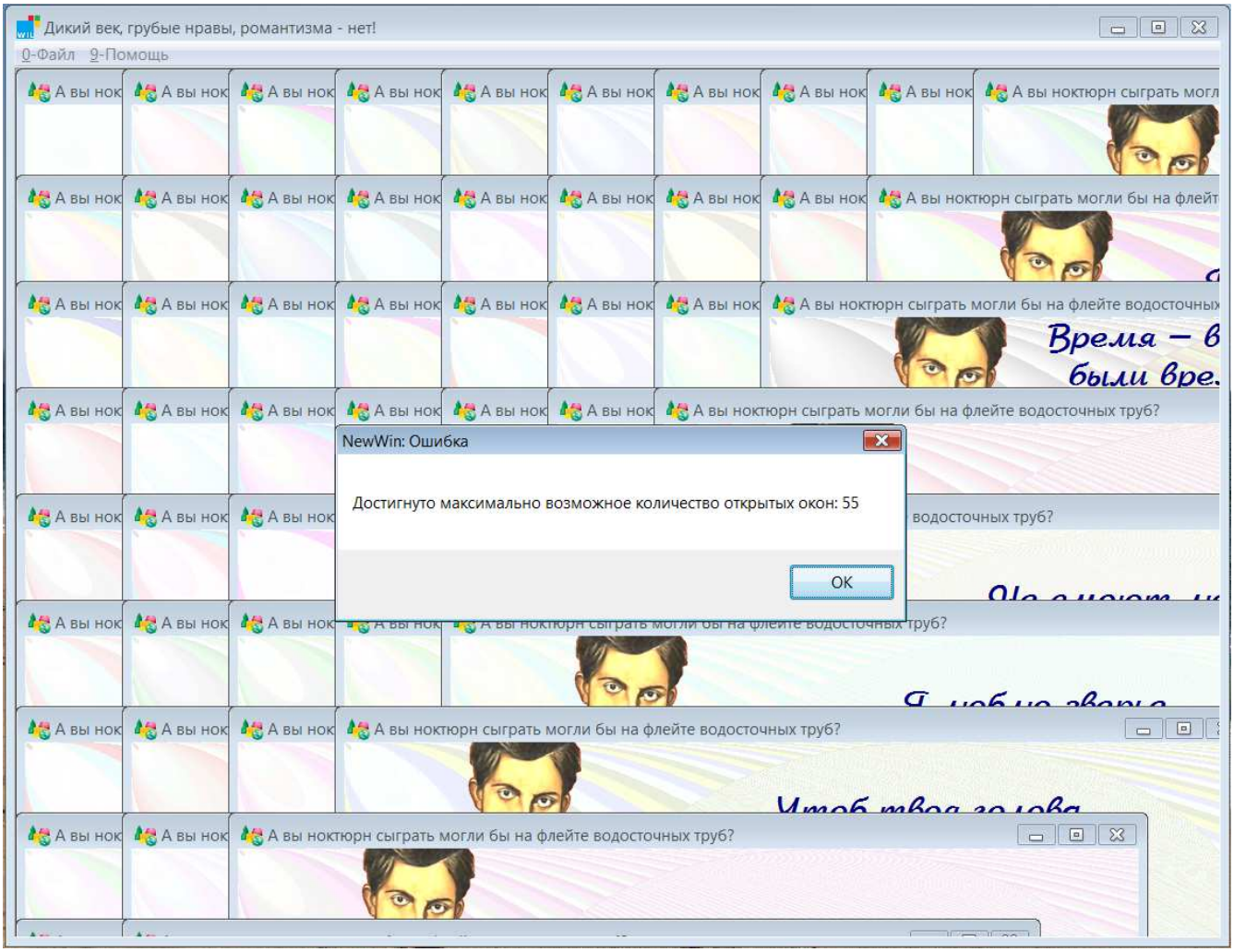


Рис. 4. Максимальное количество окон плагина «NewWin.dll».

Добавим теперь плагин «About.dll». Результат его работы виден на рис. 5.

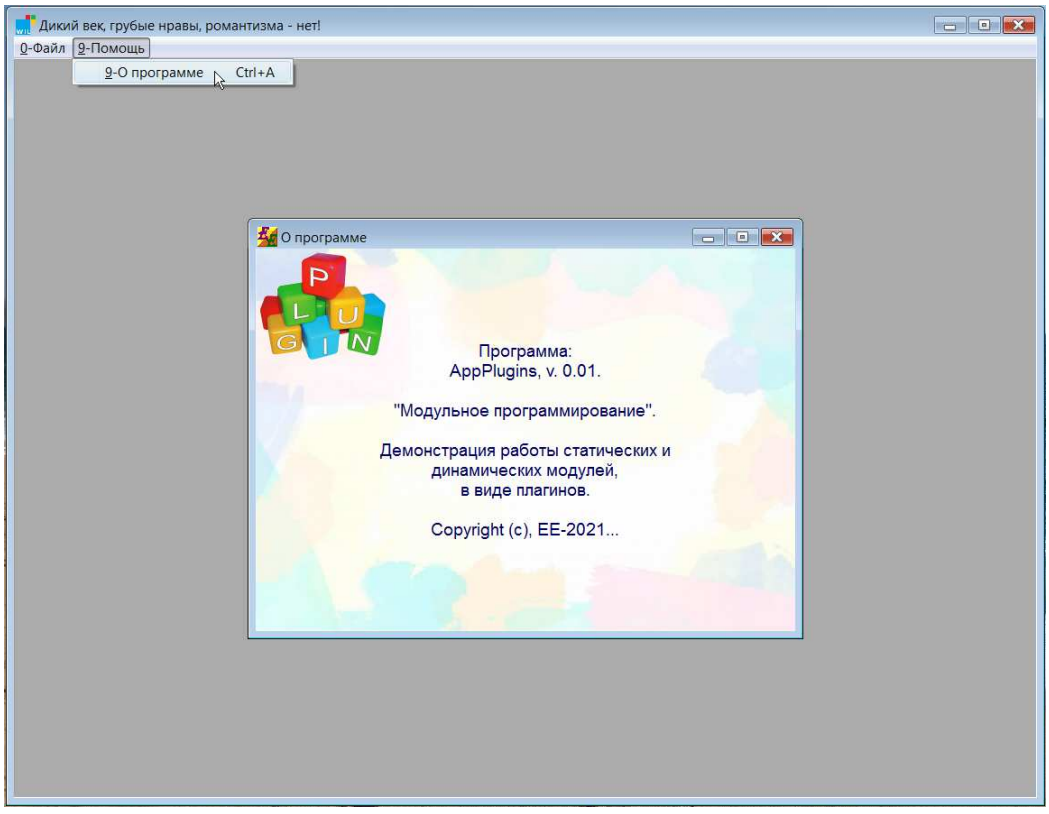


Рис.5. Единственное окно «О программе» плагина «About.dll».

Поскольку это обычное окно, а не диалог, то оно может менять фокус, размеры и местоположение. При этом размеры окна сохраняются, но, при открытии, оно всегда центрируется.

Благодаря наличию внешних ресурсов у плагинов, все эти окна полностью настраиваются. Можно менять практически все, вплоть до пиктограмм, пунктов меню, «горячих» и Alt-клавиш для них. Не говоря уже о файлах изображений и тексте в окнах.

Заметим, что для Alt-клавиш мы использовали цифры, вместо символов. Это связано с раскладкой клавиатуры. Если, допустим, раскладка у нас латинская, а буквы русские либо наоборот, то Alt-клавиши действовать не будут. Цифры, в этом смысле, от раскладки не зависят. Однако латинские символы в файловых настройках указывать можно. О самих конфигурационных файлах речь ниже.

Принцип работы программы «AppPlugins»

Здесь можно отметить, что:

1. Приложение и плагины считывают исходные данные (размеры и наименование окон, пути меню, горячие и Alt-клавиши, текстовку и т.п.) и по ним строят окна и модифицируется меню. Если данных нет, то используются параметры по умолчанию. В качестве встроенного шрифта выбран «Arbat-Bold». Если его нет в вашей системе, то используется другой.

2. При выходе из программы, текущие данные сохраняются в упрощенных ini-файлах.

Отметим, что мы практически не пользуемся rc-файлами ресурсов вообще, поскольку все ресурсы у нас динамические. В том числе и пиктограммы окон и фоновый рисунок дочернего окна. Кстати, окна можно закрывать дополнительной горячей клавишей «Esc» (помимо стандартной комбинации «Ctrl+F4»). Однако эти ресурсы должны находиться в соответствующих каталогах, относительно исполняемого exe-файла. Исключение составляет иконка для exe-файла. Поскольку, просто непонятно, как ее прикрепить к результирующему файлу, во время компиляции проекта, без использования файла ресурсов. Все остальное можно загрузить в рантайме.

Проблемы модульности

Разделять код на логически независимые модули достаточно сложно. В первую очередь, это вызвано сильной связью между используемыми программными компонентами, что хорошо видно даже для простейшего случая.

Действительно, на самом нижнем уровне, разные компоненты приложения используют общие ресурсы, которые нужно разделять. Но, собственно, сильную связь между программами определяют взаимные вызовы прикладных функций между модулями программы.

При этом, целью модульности является не столько разрыв связей между программными компонентами, сколько их унификация и стандартизация. Обычно это достигается с помощью протоколов и интерфейсов.

Общие ресурсы

В общие ресурсы входят:

1. Общие переменные (внутренние, создаваемые в памяти, и внешние, загружаемые из файлов инициализации).

2. Общее меню.

3. Общий цикл сообщений, в главном программном модуле, который обрабатывает, также, события от других модулей.

4. Общие функции и интерфейсы.

Общие переменные можно представить в виде структур, которые можно передавать по ссылке либо через указатель из одного модуля в другой. В качестве глобальных переменных можно оставить только (статические) константы.

Для целей модульности, общее меню уже нелогично делать монолитным, определяемом на уровне компиляции, в файле ресурсов. Более предпочтителен вариант, когда сам модуль добавляет нужные пункты меню, их обработчики и горячие клавиши.

Для третьего и четверного пунктов надо, первоначально, зарегистрировать плагины, т.е., получить указатели на классы (интерфейсы) внешних либо внутренних модулей и каждому из них поставить в соответствие идентификатор команды, такой же, как и для пункта меню, выполняющий этот модуль. А указатели на общие интерфейсы можно просто передавать в полях соответствующей структуры.

В общем цикле сообщений можно выделить диапазон команд (событий) которые по их номеру вызывают соответствующие регистраторы окон плагинов и их обработчики. Т.е., сделать так, чтобы эта часть работы программы не зависела от конкретного плагина, только от номера его интерфейса и стандартных функций в нем.

Часть вторая. Техническая
Общие цели

Задача, при создании нового проекта, была простой. Нужно было сразу организовать проект так, чтобы весь существенный код был вынесен во внешние бинарные модули. При этом, dll позднего связывания (динамические плагины) должны быть независимыми от приложения. Т.е., если они есть, то программа их «подхватывает» и использует, а если нет, то это никак не должно отражаться на работе основного приложения.

Кроме того, требуется, чтобы, заранее известные и необходимые модули (создание главного окна, загрузчик динамических плагинов, модуль общих функций и т.п.) организовать в виде dll раннего связывания (статические плагины). Они, конечно, являются обязательными (при полной динамической сборке проекта), но, способные заменяться на аналогичные, при наличии таковых.

При этом, в случае создания либо отладки отдельного модуля, можно весь проект подвергнуть полной статической сборке, что не должно вызывать лишних трудностей, либо ограничиться только частичной статической сборкой, для искомого и главного модуля, а для остальных – динамической сборкой.

Такой подход позволяет добиться распараллеливания сложного проекта на отдельные составляющие (как динамические, так и статические), что само по себе также является нашей целью.

Естественно, в начальной статье, речь можно вести только о самых простых случаях. А дальнейший процесс развития проекта, отражать уже в других статьях.

В подобной трактовке нет ничего особо нового, все уже давно известно и с успехом применяется на практике. Достаточно посмотреть многие открытые проекты, на том же Гитхабе, например. Однако найти подходящий прототип, на эту тему, для начинающих разработчиков, не удалось. Поэтому, цель статьи предоставить такой прототип. Который, в последствии, можно совершенствовать и развивать, в т.ч., благодаря конструктивной критике.

Мы сейчас не ведем речь о кроссплатформенности, оптимальном выборе компилятора и т.п. Для проверки общих идей это все не принципиально. Более того, чтобы не загромождать начальный проект

избыточной сложностью и детализацией, предлагается ограничиться, в демонстрационной программе, следующими задачами:

1. Проект писать на C++ / WinAPI, как для 32-х, так и для 64-х разрядных систем Windows. При этом использовать бесплатную версию Visual Studio C++ 2017, Community Edition (требуется только регистрация). Хотя сама программа должна работать и в Visual Studio C++ 2010.

2. Приложение должно представлять из себя простое главное окно, способное поддерживать статические и динамические плагины. Главными для нас являются динамические плагины. Для демонстрации их работы вполне достаточно пары штук, каждый из которых создает и поддерживает собственный тип дочернего окна.

3. Поскольку, пока нет смысла делать плагины слишком сложными, то ограничимся следующими возможностями:

3.1. Первый плагин, который мы назовем «NewWin.dll» будет создавать множество дочерних MDI окон. Каждое из которых будет выводить некий логотип, рисовать какой-нибудь фон и писать определенный текст. Все параметры должны быть настраиваемыми, храниться в конфигурационном файле и использовать внешние ресурсы. Для определенности, мы выбрали, для логотипа – фото и подпись поэта Владимира Маяковского. Для текста – его стихотворные цитаты. Цветовая гамма для фона окна выбирается случайно, а изображение строится по некоторому произвольному алгоритму (рис. 2-3).

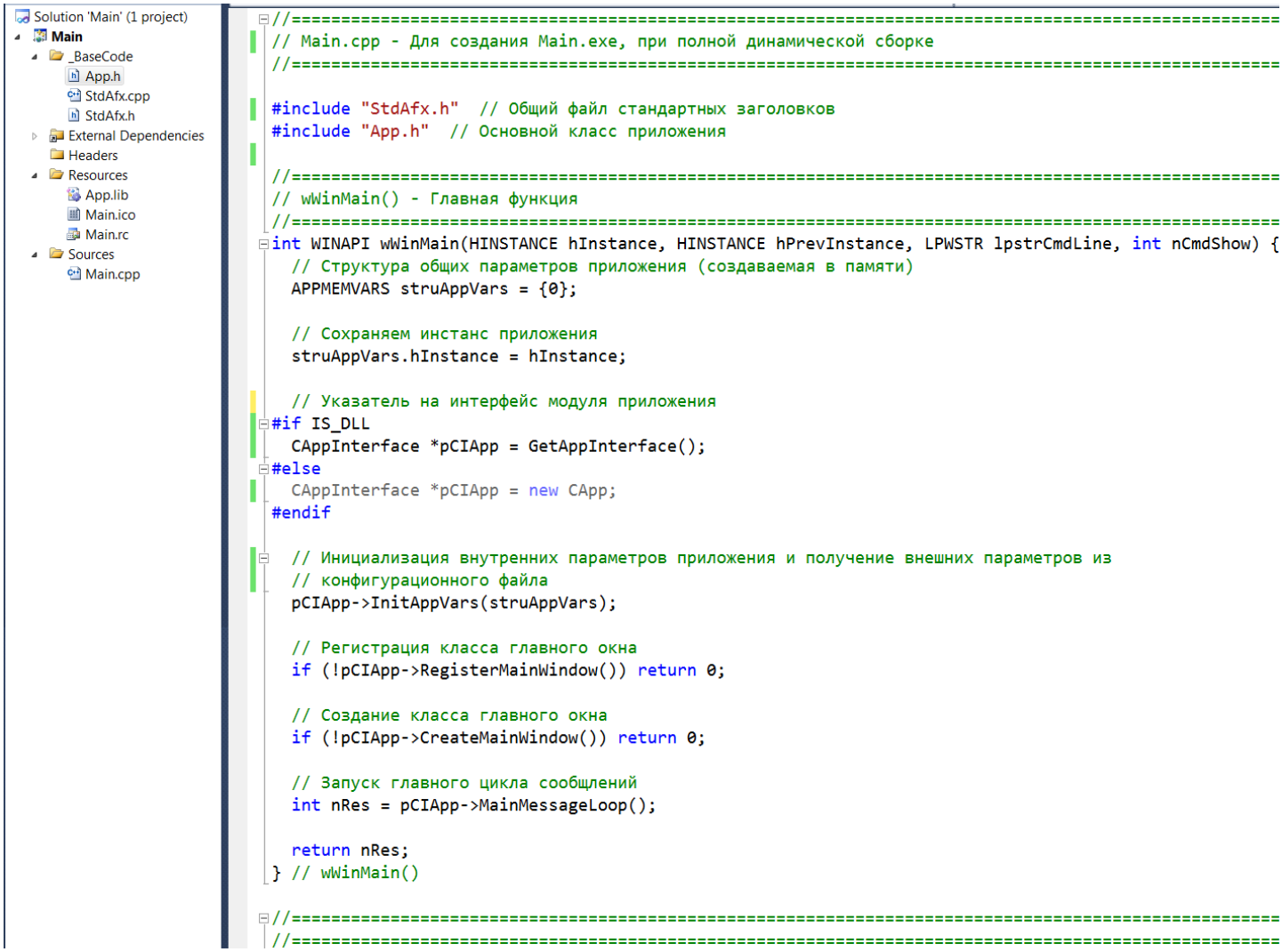
3.2. Второй плагин будет эмулировать стандартный диалог: «О программе». Это будет как бы упрощенный вариант первого плагина, позволяющий выводить только одно окно. А вместо программно генерируемого фона, там просто выводится внешний фоновый рисунок.

3.3. Каждый плагин добавляет свою запись в главное меню, «горячую» и Alt-клавишу для нее. Дополнительно (к команде «Ctrl+F4»), для MDI-окон должна действовать, как уже упоминалось выше, клавиша «Esc». Стандартная команда выхода из приложения: «Alt+F4» (ее программировать не надо). Но для нее будет доступен эквивалент в виде выбранной Alt-клавиши (рис. 1).

4. Для фиксированных dll раннего связывания (статических плагинов) будет реализован только минимально необходимый обслуживающий код.

Решение поставленной задачи методом «от обратного»

Посмотрим, каким может быть проект для главного модуля, в виде exe-файла, при полной динамической сборке (рис. 6). При этом два динамических плагина и три статических компилируются, в виде dll, независимо. Не нужно забывать об использовании необходимых lib-файлов (указанных в проектах). Последовательность компиляции, при полной динамической сборке, произвольная, но модуль «App.dll» компилируется предпоследним. Последним создается соответствующий исполнимый exe-файл. Для полной статической сборки создается только exe-модуль.




```
//=============================================================================================================
// Main.cpp - Для создания Main.exe, при полной динамической сборке
//=============================================================================================================

#include "StdAfx.h"  // Общий файл стандартных заголовков
#include "App.h"  // Основной класс приложения

//=============================================================================================================
// wWinMain() - Главная функция
//=============================================================================================================
int WINAPI wWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPWSTR lpstrCmdLine, int nCmdShow) {
  // Структура общих параметров приложения (создаваемая в памяти)
  APPMEMVARS struAppVars = {0};

  // Сохраняем инстанс приложения
  struAppVars.hInstance = hInstance;

  // Указатель на интерфейс модуля приложения
#if IS_DLL
  CAppInterface *pCIApp = GetAppInterface();
#else
  CAppInterface *pCIApp = new CApp;
#endif

  // Инициализация внутренних параметров приложения и получение внешних параметров из
  // конфигурационного файла
  pCIApp->InitAppVars(struAppVars);

  // Регистрация класса главного окна
  if (!pCIApp->RegisterMainWindow()) return 0;

  // Создание класса главного окна
  if (!pCIApp->CreateMainWindow()) return 0;

  // Запуск главного цикла сообщлений
  int nRes = pCIApp->MainMessageLoop();

  return nRes;
} // wWinMain()

//=============================================================================================================
//=============================================================================================================
```

Рис.6. Проект для главного модуля, при полной динамической сборке.

Отметим, что в папка «_BaseCode» является общей для всех проектов, независимо от способа их компиляции.

Как видим, здесь минимум кода (в файлах *.cpp). А «StdAfx.cpp» вообще пустой. Главными здесь являются файлы App.h и App.lib, которые связывают, на этапе компиляции, файлы Main.exe и App.dll.

Сам «Main.cpp» делает достаточно понятные вещи:

1. Получает указатель на интерфейс модуля приложения (либо из App.dll, либо из класса App.h / App.cpp).

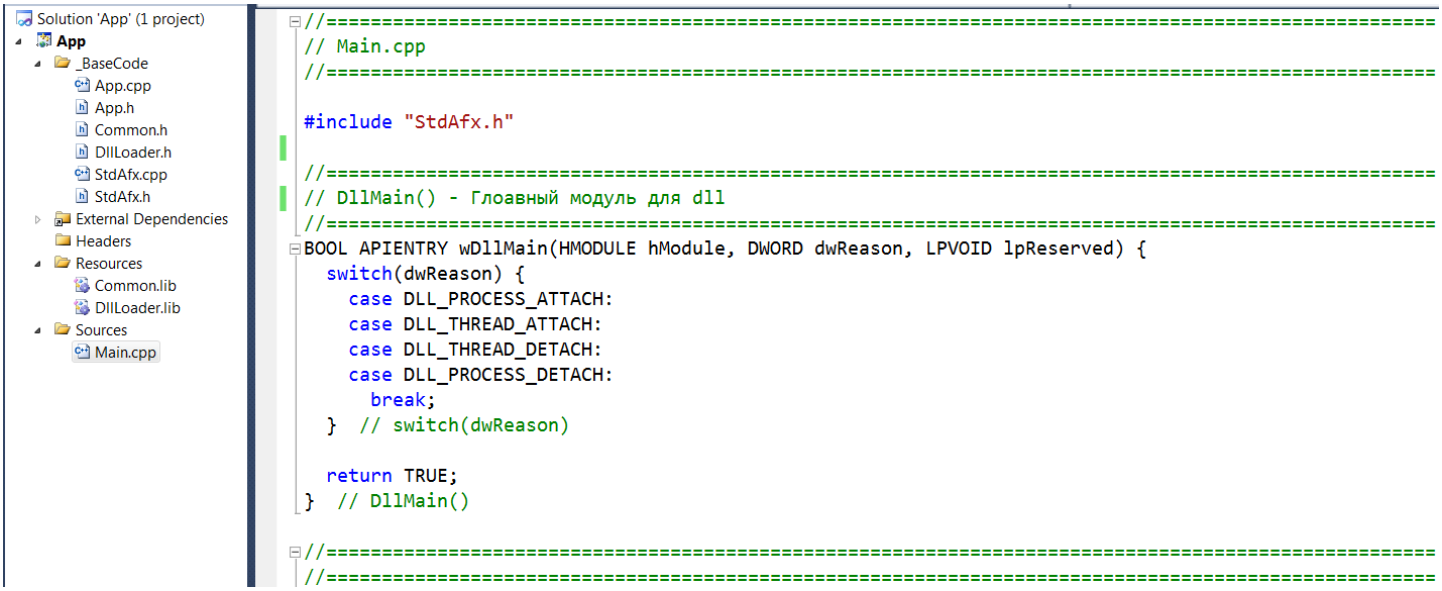
2. Создает структуру общих параметров, записывает туда инстанс главного модуля и передает ее модулю приложения для дальнейшей инициализации, в т.ч., внешними параметрами, и использования. В принципе, инстанс можно вычислить и автономно, в самой dll, так что процедура эта скорее символическая.
3. Регистрация и создание главного окна приложения.

4. Запуск цикла сообщения главного окна.

Думаю, что здесь все просто и понятно. Вопрос только, что из себя представляет интерфейс модуля приложения?

Интерфейс модуля приложения

Откроем наш проект «App» (рис. 7).




```cpp
//=============================================================================================================
// Main.cpp
//=============================================================================================================

#include "StdAfx.h"

//=============================================================================================================
// DllMain() - Глоавный модуль для dll
//=============================================================================================================
BOOL APIENTRY wDllMain(HMODULE hModule, DWORD dwReason, LPVOID lpReserved) {
  switch(dwReason) {
    case DLL_PROCESS_ATTACH:
    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
    case DLL_PROCESS_DETACH:
      break;
  }  // switch(dwReason)

  return TRUE;
}  // DllMain()

//=============================================================================================================
//=============================================================================================================
```

Рис.7. Проект «App», при полной динамической сборки модуля приложения.

Здесь вообще все просто. Код «Main.cpp» абсолютно стандартный для dll. Соответственно, главная содержательная часть находится в файлах «App.h» и «App.cpp». При этом данный модуль использует интерфейсы других dll: «Common.dll» и «DllLoader.dll», на что указывают файлы: «Common.h» / «Common.lib» и «DllLoader.h» / «DllLoader.lib». При этом все бинарные dll-модули компилируются независимо (как упоминалось выше, их у нас пять)

Посмотрим файл «App.h» из этого проекта. Там нас интересует класс «CAppInreface» (рис. 8-9).

```cpp
//=============================================================================================================
// App.h - Основной класс приложения
//=============================================================================================================

#pragma once

#include "Common.h"  // Общий файл, для приложения и плагинов
#include "DllLoader.h"  // Загрузчик плагинов

#if IS_DLL
  // Динамическая сборка (интерфейсы получаем из загруженных, во время run time, плагинов)
#else
  // Статическая сборка (интерфейсы создаем явно из этих классов)
  #include "NewWin.h"
  #include "About.h"
#endif

// Прототип сервисных функций (в данном проекте имеет символическое значение)
typedef void (*SERVICEFUNC)(APPMEMVARS &struAppVars, WSAFILEVARS wsaFileVars);

// Отображение для всех хэндлов используемых субменю в меню приложения
typedef map<wstring, HMENU> MENUPATHMAP;
```

Рис.8. Заголовки и определения в файле «App.h».

```
//=================================================================================================
// class CAppInterface - Виртуальный класс для работы с функциями главного модуля приложения
//=================================================================================================
class CAppInterface {
  //-------------------------------------------------------------------------------------------------
  // Публичные функции
  //-------------------------------------------------------------------------------------------------
  public:
    // Инициализация внутренних параметров приложения и получение внешних параметров из файла
    virtual void InitAppVars(APPMEMVARS &struAppVars) = 0;

    // Регистрация класса главного окна
    virtual BOOL RegisterMainWindow() = 0;

    // Создание класса главного окна
    virtual BOOL CreateMainWindow() = 0;

    // Запуск главного цикла сообщлений
    virtual int MainMessageLoop() = 0;

  //-------------------------------------------------------------------------------------------------
  // Приватные функции
  //-------------------------------------------------------------------------------------------------
  private:
    // Динамическое создание меню
    virtual HMENU CreateAppMenu(wstring &wsMenuElem, int nIndex=SERVICES_BASE, HMENU hPrevMenu=NULL) = 0;
};  // class CAppInterface
```

Рис.9. Класс «CAppInreface» в файле «App.h».

Это типичное оформление для виртуального класса, который мы перегружаем в классе «CApp» (рис. 10).

```
//=============================================================================================================
// class CApp - Класс для работы с приложением
//=============================================================================================================
class CApp : public CAppInterface {
  //-------------------------------------------------------------------------------------------------------
  // Публичные функции
  //-------------------------------------------------------------------------------------------------------
  public:
    // Инициализация внутренних параметров приложения и получение внешних параметров из файла
    virtual void InitAppVars(APPMEMVARS &struAppVars) override;

    // Регистрация класса главного окна
    virtual BOOL RegisterMainWindow() override;

    // Создание класса главного окна
    virtual BOOL CreateMainWindow() override;

    // Запуск главного цикла сообщлений
    virtual int MainMessageLoop() override;

  //-------------------------------------------------------------------------------------------------------
  // Приватные функции
  //-------------------------------------------------------------------------------------------------------
  private:
    // Динамическое создание меню
    virtual HMENU CreateAppMenu(wstring &wsMenuElem, int nIndex=SERVICES_BASE, HMENU hPrevMenu=NULL) override;

  //-------------------------------------------------------------------------------------------------------
  // Собственные приватные функции
  //-------------------------------------------------------------------------------------------------------
  private:
    // Получить и сохранить размеры и позицию главного окна приложения
    static void GetAppWindowPos(HWND hWnd);

    // Запись параметров приложения в ini-файл и выход из программы
    static void AppExit(APPMEMVARS &struAppVars, WSAFILEVARS wsaFileVars);

    // Оконная функция главного окна приложения
    //virtual LRESULT CALLBACK MainWndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam) override;
    static LRESULT CALLBACK MainWndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam);

  //-------------------------------------------------------------------------------------------------------
  // Данные
  //-------------------------------------------------------------------------------------------------------
  private:
    // Массив текстовых наименований для полей структуры внешних (файловых) параметров приложения
    static wstring m_wsaComments[];  // g_eMax + 1

    // Массив строковых значений, по умолчанию, для полей структуры внешних (файловых) параметров приложения
    static wstring m_wsaFileVars[];  // g_eMax + 1

    // Параметры приложения
    static APPMEMVARS m_struAppVars;  // Структура общих (глобальных) параметров приложения (создаваемая в памяти)
    //static WSAFILEVARS m_wsaFileVars;  // Массив файловых параметров (полученный из главного ini-файла)
    static LPCWSTR m_wsMainClassName;  // Имя создаваемого класса главного окна
    static LPCWSTR m_wsMainLogFile;  // Наименование программного лог-файла
    static LPCWSTR m_wsMainIniFile;  // Наименование главного ini-файла
    static LPCWSTR m_wsMainIcon;  // Большая иконка  (во внешнем, относительно плагина, каталоге ресурсов)
    static HACCEL m_hAccelTable;  // Загрузить таблицу акселераторов

    // Сервисные функции
    static SERVICEFUNC ServiceFunctions[];  // Массив сервисных функций
    static const int m_nMaxServiceCmd;  // Общее количество сервисных функций в массиве (вычисляется)

    // Меню
    static MENUPATHMAP MenuPathMap;  // Отображение путей меню / субменю на их хэндлы

    // Указатель на особый (общий) интерфейс
    static PICOMMON m_pICommon;  // Указатель на особый (общий) интерфейс плагина Common.dll

    // Указатель на загрузчик плагинов
    static CDllLoaderInterface *m_pDllLoader;

    // Указатели на динамические интерфейсы
    static PIPLUGINVEC m_Plugins;  // Вектор указателей интерфейсов динамических плагинов (dll)

    // Параметры плагинов
    static LPCWSTR m_wsPlugPathMask;  // Маска для используемых плагинов
};  // class CApp
```

Рис.10. Перегруженный класс «CApp» в файле «App.h» для модуля приложения.

Здесь же можно найти и прототип экспортируемой функции «GetAppInterface()» (рис. 11).

```
#if IS_DLL
  //=============================================================================================================
  // GetAppInterface() - Это экспортированная функция, которая определяется в cpp-модуле
  //=============================================================================================================
  extern "C" DLL_API CAppInterface *GetAppInterface();
#endif
```

Рис. 11. Прототип экспортируемой функции «GetAppInterface()».

А ее реализация находится в файле «App.cpp» (рис. 12):

```
#if IS_DLL
 //=============================================================================================================
 // GetAppInterface() - Это экспортированная функция, которая будет вызываться главным модулем
 //=============================================================================================================
 extern "C" DLL_API CAppInterface *GetAppInterface() {
   // Указатель на общий интерфейс
   static CAppInterface *m_pAppInterface = new CApp;  // Указатель на статический класс CApp

   return m_pAppInterface;
 }  // GetAppInterface()
#endif
```

Рис. 12. Реализация экспортируемой функции «GetAppInterface()».

Как видим, в принципе, все относительно просто. Сложны только нюансы реализации. Например, в нашем перегруженном классе «CApp» добавлены собственные функции и множество различных данных. Со всем этим надо работать, что требует много кода.

Однако, концептуально, виртуальные интерфейсы у нас одни и те же, как для статических плагинов, так и для динамических.

В любом случае, чтобы разобраться с проектом, надо с ним поработать. Как говориться: «Лучше один раз потрогать, чем сто раз увидеть» :) .

Нюансы реализации

Их достаточно много, даже для этого небольшого проекта. Я бы обратил внимание на:

1. Логирование. Механизм здесь этот есть, но «замороженный». Я использовал его, когда надо было отследить маршруты оконных сообщений в циклах сообщений плагинов и главного окна.

2. Работа с ini-файлами. Оказалось, что нет особой необходимости работать с полноценными ini-файлами. Для этого надо много кода, который, для наших целей, вполне можно упростить. На рис. 13, показан пример файла «About.ini», генерируемого по умолчанию.

```
; Заголовок окна плагина:
О программе
; Ширина окна плагина (в промилле от клиентской области родителя):
449
; Высота окна плагина (в промилле от клиентской области родителя):
484
; Путь меню для обработчика комманды и ее хоткей:
&9-Помощь/&9-О программе  Ctrl+A
; Текст для окна плагина:
Программа:
AppPlugins, v. 0.01.

"Модульное программирование".

Демонстрация работы статических и
динамических модулей,
в виде плагинов.

Copyright (c), EE-2021...
```

Рис. 13. Файл «About.ini», генерируемый по умолчанию.

Структура его очень простая, но всегда фиксированная, для использования в конкретном модуле:

– Строка комментария.
– Одна или несколько строк данных.

Строка комментария всегда начинается с символа ';'. А строка / строки данных это все, что лежит после текущего комментария и до следующего комментария, либо конца файла.

Это позволило оформить достаточно просто громоздкие данные для плагина «NewWin.dll» (в файле «NewWin.ini» (рис. 14).

```
; Заголовок окна плагина:
А вы ноктюрн сыграть могли бы на флейте водосточных труб?
; Ширина окна плагина (в промилле от клиентской области родителя):
848
; Высота окна плагина (в промилле от клиентской области родителя):
548
; Путь меню для обработчика комманды и ее хоткей:
&0-Файл/&0-Новое окно Ctrl+N
; Общее количество тестовых блоков для окон:
12
; Текст No. 01 для окна плагина:
Время – вещь необычайно длинная, –
были времена – прошли былинные.
Ни былин, ни эпосов, ни эпопей.
Телеграммой лети, строфа!
Воспаленной губой припади
и попей из реки по имени – «Факт».
Это время гудит телеграфной струной,
это сердце с правдой вдвоем.
Это было с бойцами, или страной,
или в сердце было в моем.
; Текст No. 02 для окна плагина:
Будет луна.
Есть уже немножко.
```

Рис. 14. Часть файла «NewWin.ini», генерируемого по умолчанию.

Если изменить здесь данные, без нарушения их структуры, то программа соответствующим образом отреагирует.

Заметим, что для отделения команды хоткея, используется символ табулятора '\t'. Кроме того, вместо процентов мы используем промилле (тысячную долю единицы), а вместо абсолютных размеров относительные. При желании это все можно поменять.

Кстати, все параметры у нас строковые, но поскольку их структура для каждого модуля нам известна, что числа из строк и обратно получаются соответствующими преобразованиями. При этом сам код чтения / записи наших ini-файлов, достаточно простой.

3. У нас используется достаточно много констант, для данных по умолчанию, определяемых в начале *.cpp файлов. Например, верхняя часть «App.cpp» показана на рис. 15-16.

```
//=================================================================================================
// Внутренние константы (для инициализации значений по умолчанию)
//=================================================================================================

// Параметры приложения
LPCWSTR CApp::m_wsMainClassName = L"MainWindow";  // Имя создаваемого класса главного окна
//LPCWSTR CApp::m_wsMainLogFile = L"App\\Main.log";  // Наименование программного лог-файла
LPCWSTR CApp::m_wsMainLogFile = L"";  // Трюк, чтобы не создавать лог-файл!
LPCWSTR CApp::m_wsMainIniFile = L"App\\Main.ini";  // Наименование главного ini-файла
LPCWSTR CApp::m_wsMainIcon = L"App\\Main.ico";  // Большая иконка (во внешнем каталоге ресурсов)
HACCEL CApp::m_hAccelTable = NULL;  // Загрузить таблицу акселераторов
//LPCWSTR CApp::m_wsMenuName = L"MAINMENU";  // Имя меню (в файле ресурсов)
//LPCWSTR CApp::m_wsAccelTable = L"MAINACCS";  // Имя таблицы акселераторов (в файле ресурсов)

// Параметры плагинов
LPCWSTR CApp::m_wsPlugPathMask = L"Plugins\\*.dll";  // Маска для используемых плагинов

//=================================================================================================
// Глобальное перечисление наименований для структуры внешних (файловых) параметров приложения
//=================================================================================================
enum {
  g_eWinTitle = 0,  // Заголовок главного окна приложения
  g_eWinWidth1000,  // Ширина окна приложения (в промилле от рабочей области)
  g_eWinHeight1000,  // Высота окна приложения (в промилле от рабочей области)
  //g_eWinLeft1000, // Отступ слева окна приложения (в промилле от рабочей области)
  g_eWinTop1000, // Отступ сверху окна приложения (в промилле от рабочей области)
  eCommand,  // Путь меню для обработчика комманды и ее хоткей
  g_eMax  // Последний (максимальный) элемент структуры внешних параметров приложения
};  // enum
```

Рис. 15. Константы и глобальное перечисление, для файла «App.cpp».

```
//=============================================================================================================
// Внутренние константы (для инициализации значений по умолчанию)
//=============================================================================================================

// Массив текстовых наименований для структуры внешних (файловых) параметров плагинов
wstring CApp::m_wsaComments[g_eMax + 1] = {
  L"; Заголовок главного окна приложения:",  // g_eWinTitle
  L"; Ширина окна приложения (в промилле от рабочей области):",  // g_eWidth1000
  L"; Высота окна приложения (в промилле от рабочей области):",  // g_eHeight1000
  //L"; Отступ слева окна приложения (в промилле от рабочей области):",  // g_eLeft1000
  L"; Отступ сверху окна приложения (в промилле от рабочей области):",  // g_eTop1000
  L"; Путь меню для обработчика комманды и ее хоткей:",  // eCommand
  L""  // g_eMax - Последний (максимальный) элемент структуры внешних параметров
};  // CApp CNewWin::m_wsaComments[g_eMax + 1]

// Массив строковых значений, по умолчанию, для структуры внешних (файловых) параметров плагинов
wstring CApp::m_wsaFileVars[g_eMax + 1] = {
  L"Дикий век, грубые нравы, романтизма - нет!",  // g_eWinTitle
  L"600",  // g_eWidth1000
  L"850",  // g_eHeight1000
  //L"0",  // g_eLeft1000
  L"60",  // g_eTop1000
  L"&0-Файл/&9-Выход\tAlt+F4",  // eCommand
  L""  // g_eMax - Последний элемент структуры внешних параметров
};  // wstring CApp::m_wsaFileVars[g_eMax + 1]
```

Рис. 16. Массивы данных, по умолчанию, для файла «App.cpp».

Это далеко не все имеющиеся нюансы реализации, но обо всем упоминать довольно долго. Лучше, не полениться и изучить прилагаемый код.

Статическая сборка

Поскольку весь существенный код у нас находится в каталоге «_BaseCode», то для полной статической сборки достаточно его подключить и убрать флаг **IS_DLL** (рис. 17).

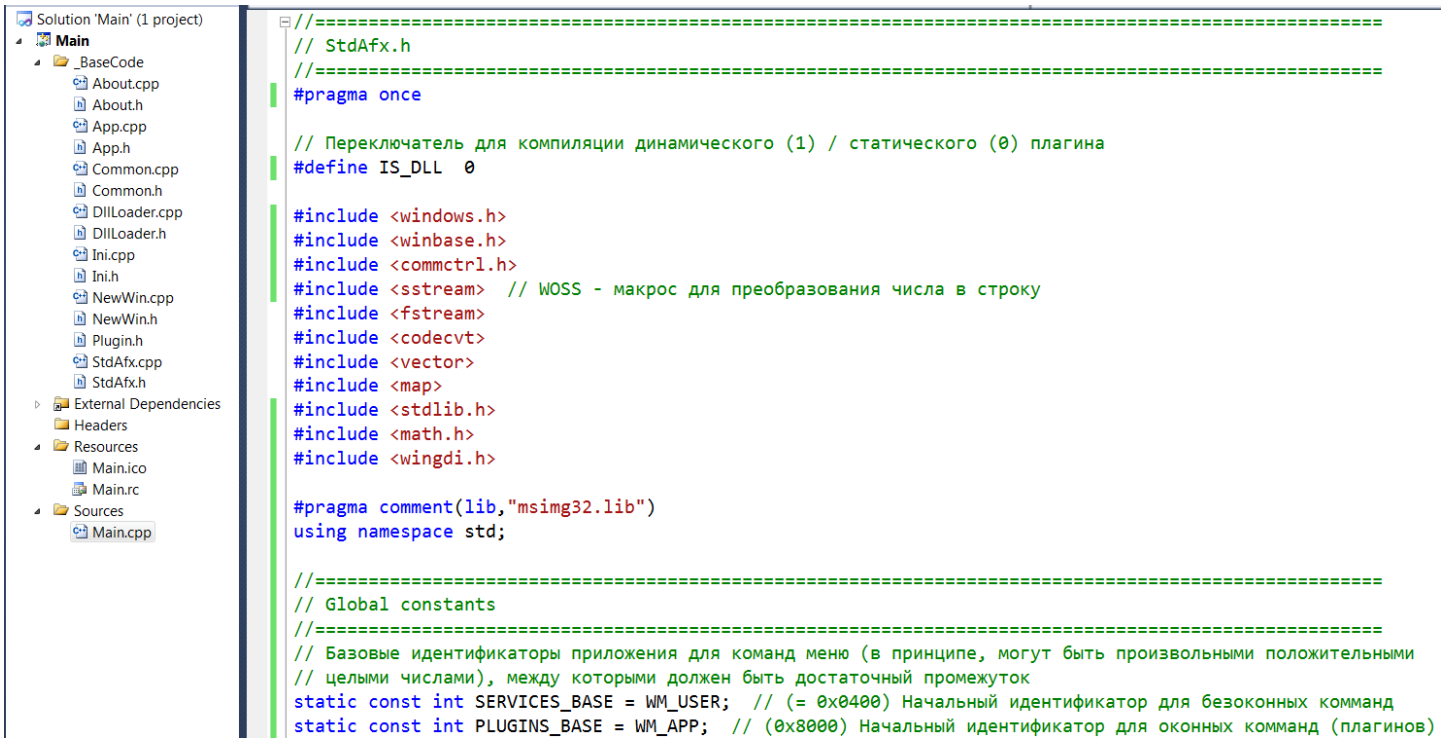


Рис.17. Проект для главного модуля, при полной статической сборке.

Никакого другого изменения кода в наших проектах мы не делаем. Но не надо забывать, что проекты для статической и динамической сборки у нас хотя и разные, однако у них общий файл «StdAfx.h». Поэтому всегда нужно проверять, какое значение параметра **IS_DLL** выставлено там.

Все, компилируем и получаем exe-шники, для x86 и x64, независящие от наших dll. Результаты всей нашей работы можно посмотреть, например, в TotalCommander'e (рис. 18). Имена для exe-файлов можно указывать любые.

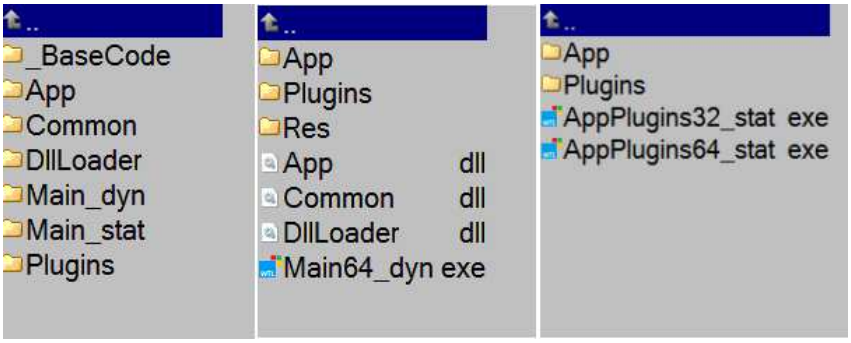


Рис.18. Каталоги проекта для динамической и статической сборки.

Где скачать проект

Файлы проекта можно скачать по адресу:
http://emery-emerald.narod.ru/Articles/AppPlugins/AppPlugins_src.zip

Бинарные файлы, полученные в результате компиляции (для 32-х и 64-х разрядных систем) можно получить здесь:
http://emery-emerald.narod.ru/Articles/AppPlugins/AppPlugins_bin.zip

Pdf-версию данной статьи можно найти по ссылке:
http://emery-emerald.narod.ru/Articles/AppPlugins/AppPlugins.pdf

Компиляция под разными версиями Visual Studio C++

Данный проект скомпилирован под Visual Studio C++ 2017, Community Edition и SDK 10.0 как для x86, так и x64. Но он может быть получен и на меньших версиях и SDK 8.1, если в файлах, типа, *.vcxproj, сделать соответствующую замену для строк «ToolsVersion="15.0"» и «v141»:

– В версии VS C++ 2010: «ToolsVersion="4.0"» и «v100».
– В версии VS C++ 2013: «ToolsVersion="12.0"» и «v120».
– В версии VS C++ 2015: «ToolsVersion="14.0"» и «v140».

Часть третья. Выводы

Конечно, рассмотренная задача не слишком практичная, чтобы делать далеко идущие выводы. Скорее, это начальный прототип, для работы в данном направлении. Если будет прогресс, то можно поговорить об этом в следующих статьях.

Метки:

C++, WinAPI, статические и динамические плагины, static and dynamic plugins, dll, упрощенные ini-файлы