**Реализация отложенной загрузки библиотек на С++**

Андрей Солодовников

Вы все еще грузите библиотеки вручную?

Тогда мы идем к вам!

**Краткая предыстория**

По специфике моей работы мне довольно часто приходится вручную загружать библиотеки и динамически, при помощи GetProcAddress, импортировать множество функций. Это происходит отчасти потому, что требуется обеспечить совместимость с различными версиями Windows, в которых целевые функции могут отсутствовать, отчасти потому, что так бывает удобнее (например, при реализации механизма плагинов). Конечно, всегда хочется это автоматизировать, особенно если функций и библиотек много. С одной стороны, в линейке Visual C++ для этого есть поддержка компилятора\линкера в виде механизма Delay Load, с другой стороны, бытует мнение, что использовать этот метод является дурным тоном, и, наверное, это так. Одна из основных причин, которую хочется отметить особенно – этот механизм является microsoft-specific, то есть никаких гарантий, что написанный Вами код будет работать и на других компиляторах или платформах, нет. Более того, несколько раз «попав» на странное поведение этого механизма (например, см. Q218613), мы от его использования в своих проектах отказались.



Следующим шагом был поиск готового подходящего функционала. Как ни странно, такого не находилось, несмотря на то, что проблема действительно имеет место быть. Многие решения были слишком просты и неоптимальны (например, это решение). Они не позволяли определять импорт сразу нескольких функций из одной библиотеки, либо для этого нужно было написать приличное количество кода. Они вызывали GetProcAddress и LoadLibrary в любое время, когда им вздумается, а на самом деле – чуть ли не при каждом обращении к импортируемой функции. Другие (например, такое решение) было достаточно сложно и неудобно использовать.



|  |
| --- |
| ПРИМЕЧАНИЕ  На самом деле, указанные варианты вполне могут быть использованы в небольших проектах, когда не требуется импортировать большое количество функций. Однако их использование в любом случае требует достаточно много усидчивости и терпения, по крайней мере, меня это не устраивало. |

И общий недостаток всех этих решений – они были и есть неоптимальны. Особенно это касается количества кода, генерируемого компилятором (да и программистом) на одну импортируемую функцию и быстродействия полученного кода.

Все это, вкупе с потраченным временем, сподвигло меня к необходимости написания очередного велосипеда в виде библиотеки эмуляции Delay Load, а также и этой статьи.

**Требования к библиотеке, реализующей механизм Delay load**

В данном параграфе мы рассмотрим более подробно, каким базовым требованиям должен удовлетворять механизм динамической загрузки библиотек.

Исходя из описанного выше, можно сформулировать следующие требования к механизму поддержки динамической загрузки библиотек:

Как можно большая независимость от компилятора С++ (в пределах ANSI C++). Минимальные требования к компилятору – библиотека должна быть полностью функциональна на всех Visual C++ компиляторах, начиная с Visual C++ 6.0;

Минимальное количество кода, генерируемого компилятором, которое приходится на одну импортируемую функцию;

**Удобство определения в проекте импортируемых библиотек\функций;**

Возможность задания своих стратегий (реакций) на ошибки загрузки библиотеки\нахождения функции;

Минимизация вызовов LoadLibrary. Для одной библиотеки (модуля) вызов LoadLibrary должен производится один раз вне зависимости от количества импортируемых из нее функций. Данный механизм должен работать не только в пределах одной единицы трансляции, но и проекта в целом. Таким образом, должна создаваться единая для приложения таблица используемых модулей;

Минимизация вызовов GetProcAddress. GetProcAddress должен вызываться только при первом обращении к импортируемой функции, в дальнейшем все вызовы импортируемой функции должны производиться напрямую;

Библиотека должна обеспечивать привычный синтаксис вызова – не должно быть никаких внешних отличий от обычного вызова функции из С/С++;

Должны поддерживаться UNICODE версии импортируемых функций, причем, желательно, на основе заголовков от соответствующих статически линкуемых библиотек, в которых определены соответствующие макросы.

Из описанных выше требований наиболее важными и интересными представляются пункты 3,5,6 и 7. Особенности их реализации будут рассмотрены более подробно далее вместе с программной реализацией библиотеки. Для тех, кому детали реализации не интересны, а интересны методы использования библиотеки, предлагается приступать сразу к разделу Использование библиотеки.

Предлагаемая реализация библиотеки

Класс, инкапсулирующий работу с модулями

Для начала рассмотрим требование (5) к реализации загрузки библиотек\модулей. Очевидно, что обеспечить выполнение данного требования с условием уникальности экземпляра библиотеки в пределах всех единиц трансляции проекта можно использованием паттерна Singlton для загружаемого модуля. При этом для каждого различного загружаемого модуля должен создаваться собственный экземпляр синглтона, который и будет обеспечивать «одноразовую» загрузку в конструкторе и в дальнейшем выгрузку библиотеки в деструкторе. Эта задача решается определением шаблонного класса CModule. Имя библиотеки должно служить значением, относительно которого производится инстанцирование объекта, инкапсулирующего загрузку библиотеки. Поскольку в качестве паттерна Singlton используется синглтон Мейерса, то в качестве бонуса мы получаем отложенную загрузку библиотек (поскольку создание экземпляра синглтона производится при первом обращении к порождающей функции).

|  |  |  |
| --- | --- | --- |
| СОВЕТ  Напомню, что простейшая реализация синглтона Мейерса выглядит следующим образом: | | |
| template <class T>  struct CMeyersSinglton  {  static T& GetInstance()  {  static T obj;  return obj;  }  }; |

В связи с этим первый вариант определения шаблона CModule мог бы выглядеть так:

|  |
| --- |
| template <LPCTSTR Name>  class CModule; |

Тут следует сделать небольшое отступление. Как было бы прекрасно, если бы любой абстрактный язык программирования, используемый нами, обеспечивал бы любую востребованную нами возможность. Но, очевидно, по соображениям здравого смысла, это невыполнимо, поэтому приходится пользоваться тем, что есть. А есть такая неприятная вещь – в С++ напрямую инстанцировать шаблон строковым литералом не получится. Шаблон может быть инстанцирован только константой с external linkage, а строковый литерал имеет internal linkage. На первый взгляд, все достаточно печально. Однако, как обычно, решение лежит на поверхности. Оно очень простое и очевидное. Мы будем инстанцировать шаблон модуля уникальным классом, инкапсулирующим строковый литерал. Сам же класс будет формироваться при помощи макросов:

|  |
| --- |
| #define DECLARE\_NAME\_ID\_IMPL(id, name, ret, text)\  struct NAME\_ID(id)\  {\  enum {length = sizeof(name)};\  static ret GetStr(){return text(name);}\  };  #define DECLARE\_NAME\_ID\_A(id, name) DECLARE\_NAME\_ID\_IMPL(id, name, LPCSTR, DL\_EMPTY())  #define DECLARE\_NAME\_ID(id, name) DECLARE\_NAME\_ID\_IMPL(id, name, LPCTSTR,\_T) |

Данный класс является универсальным и будет использован в дальнейшем и для представления имен импортируемых функций. Но и тут есть один маленький нюанс – поскольку функция GetProcAddress использует только ANSI строки, то мы вынуждены это предусмотреть, объявив дополнительный макрос DECLARE\_NAME\_ID\_A.

Итак, в связи со всем вышеизложенным, определение шаблона CModule без учета стратегий будет выглядеть так:

|  |
| --- |
| template <class Name>  class CModule; |

Теперь добавим стратегии загрузки\выгрузки модуля. Поскольку стратегия контролирует процессы, связанные с загрузкой и выгрузкой, у нее должно быть как минимум 2 функции. Одна отвечает за загрузку модуля, вторая за его выгрузку:

|  |
| --- |
| struct CModulePolicy  {  static HMODULE Load(LPCTSTR szFileName);  static BOOL Free(HMODULE hModule);  }; |

Теперь у нас есть все, что необходимо для полного написания класса CModule. Реализация его в предлагаемой библиотеке приведена в листинге ниже:

|  |
| --- |
| struct CModuleLoadLibraryPolicy  {  static HMODULE Load(LPCTSTR szFileName)  {  return ::LoadLibrary(szFileName);  }  static BOOL Free(HMODULE hModule)  {  return ::FreeLibrary(hModule);  }  };  struct CModuleGetModuleHandlePolicy  {  static HMODULE Load(LPCTSTR szFileName)  {  return ::GetModuleHandle(szFileName);  }  static BOOL Free(HMODULE hModule)  {  return TRUE;  }  };  template <class Name, class LoadPolicy = CModuleLoadLibraryPolicy>  class CModule  {  public:  typedef CModule<Name, LoadPolicy> type;  typedef Name name\_type;  static type &GetModule()  {  #ifdef DL\_MT  static volatile LONG lMutex = FALSE;  CLWMutex theMutex(lMutex);  CAutoLock<CLWMutex> autoLock(theMutex);  #endif //DL\_MT  static type Module;  return Module;  }  HMODULE GetModuleHandle() const  {  return m\_hModule;  }  BOOL IsLoaded() const  {  return m\_hModule != NULL;  }  // Caution - use with care. Not thread-safe  BOOL UnloadModule()  {  HMODULE hModule = m\_hModule;  m\_hModule = NULL;  return LoadPolicy::Free(hModule);  }  ~CModule()  {  if (m\_hModule)  UnloadModule();  }  private:  CModule()  {  m\_hModule = LoadPolicy::Load(name\_type::GetStr());  }  HMODULE m\_hModule;  }; |

Класс модуля позволяет явно выгружать библиотеку (модуль) при помощи функции UnloadModule, однако пользоваться этой возможностью надо с большой осторожностью.

**Реализация динамического поиска функций и глобальной таблицы импорта**

Теперь рассмотрим детали реализации пунктов 6 и 7 (поиск адресов импортируемых функций и их вызов). Это наиболее нетривиальная и интересная в плане программирования часть библиотеки, поскольку функции могут иметь различное число параметров, а также различные типы возвращаемых значений. И напомню основное требование – естественный синтаксис вызова функций и минимизация обращений к GetProcAddress.

В данном случае для обеспечения требования минимизации вызовов GetProcAddress мы будем использовать технику создания прокси-функций. Фактически, при первом вызове импортируемой функции мы будем попадать в сформированную компилятором прокси-функцию, в которой будет производиться поиск адреса функции по ее имени в библиотеке и в зависимости от успешности поиска производится либо вызов функции, либо выполнение операции, заданной в стратегии реакции на ошибки поиска. Для того, чтобы в дальнейшем вызывалась непосредственно импортируемая функция, а не прокси, адрес, полученный в прокси, запоминается в глобальной для всех единиц трансляции таблице указателей на функции. Для создания таблицы используется техника, подобная применяемой в синглтоне Мейерса. В сильно упрощенном виде это выглядит так:

|  |
| --- |
| template <class Proxy>  struct CGlobalProxyTable  {  static FARPROC &GetProxy()  {  static FARPROC proxy;  return proxy;  }  }; |

В данном примере для каждого входного типа будет сгенерирован уникальный глобальный указатель типа FARPROC, фактически являющийся ячейкой глобальной в терминах единиц трансляций таблицы функций.

Для того, чтобы определить интерфейс ячейки таблицы функций, выясним, от чего зависит импортируемая функция. Очевидно, это имя функции, модуль, из которого надо ее импортировать, и прокси, используемый для определения адреса функции в загружаемой библиотеке. В связи с этим определим класс CDynFunction, инкапсулирующий ячейку для хранения адреса функции в глобальной таблице импортируемых функций:

|  |
| --- |
| template <class Module, class Name, class Proxy>  class CDynFunction |

Учитывая все вышесказанное, реализация класса тривиальна и будет выглядеть так:

|  |
| --- |
| template <class Module, class Name, class Proxy>  class CDynFunction  {  public:  typedef CDynFunction<Module, Name, Proxy> type;  typedef Proxy proxy\_type;  typedef Module module\_type;  typedef Name name\_type;    static typename proxy\_type::fun\_type &GetProxy()  {  static typename proxy\_type::fun\_type proxy = proxy\_type::template Proxy<type>::ProxyFun;  return proxy;  }  static BOOL InitFunction()  {  #ifdef DL\_MT  static volatile LONG lMutex = FALSE;  #endif // DL\_MT  const module\_type &theModule = module\_type::GetModule();  if (theModule.IsLoaded())  return DL\_GetProcAddressImpl(  #ifdef DL\_MT  lMutex,  (const FARPROC)proxy\_type::template Proxy<type>::ProxyFun,  #endif //DL\_MT  (volatile FARPROC &)GetProxy(),  theModule.GetModuleHandle(),  name\_type::GetStr()  );  return FALSE;  }  }; |

Функция DL\_GetProcAddressImpl представляет собой обертку GetProcAddress, и вынесена в отдельный функциональный элемент для уменьшения размера кода при поддержке многопоточности. Статический метод GetProxy() вернет глобальный в смысле единиц трансляции адрес в таблице функций, причем изначально по этому адресу находится адрес прокси функции. Таким образом, вызывая функцию по указателю, полученному при помощи GetProxy(), мы первоначально вызываем прокси, а в дальнейшем будем вызывать импортируемую функцию напрямую.

**Реализация прокси функций**

До этого момента все было достаточно очевидно и довольно просто. Однако при попытке реализации класса, определяющего функционал прокси-функции, мы сталкиваемся с проблемами. Чтобы понять, в чем они заключаются, рассмотрим параметры, необходимые для генерации прокси функции. Это:

тип возвращаемого значения импортируемой функции;

список типов параметров импортируемой функции;

стратегия реакции на ошибку поиска функции в модуле;

тип ячейки глобальной таблицы указателей на импортируемые функции (CDynFunction), который будет использован при создании прокси.

Как известно, С++ не поддерживает шаблоны с переменным количеством параметров. В связи с этим придется использовать генерацию экземпляров шаблона при помощи макросов а-ля boost::preprocessor. Объяснять подробно здесь, как это работает, я не буду – это тема для отдельной статьи. Кроме того, все это удовольствие осложняется тем, что Visual C 6.0 не может возвращать из void функции тип void. Для обхода этой проблемы приходится создавать отдельные классы для «нормальных» типов и для void, а затем использовать специализацию шаблона по возвращаемому значению с последующим наследованием.

Рассмотрим реализацию, предлагаемую в библиотеке:

|  |
| --- |
| #define FUN\_PROXY(n) DL\_CAT(CFunProxy,n)  #define FUN\_PROXY\_IMPL(n) DL\_CAT(FUN\_PROXY(n),Impl)  #define DECLARE\_FUN\_PROXY(param\_count) \  template <typename R>\  struct FUN\_PROXY\_IMPL(param\_count)\  {\  template <class DynFunction, DL\_REPEAT\_N(param\_count, typename P), class Policy> struct RetProxy\  {\  static R WINAPI ProxyFun(DL\_REPEAT\_PARAM\_N(param\_count, P, v))\  {\  if (DynFunction::InitFunction())\  return DynFunction::GetProxy()(DL\_REPEAT\_N(param\_count, v));\  return Policy::template FunctionTrait<DynFunction>::MakeReturn();\  }\  };\  };\  \  template <>\  struct FUN\_PROXY\_IMPL(param\_count) <void>\  {\  template <class DynFunction, DL\_REPEAT\_N(param\_count, typename P), class Policy> struct RetProxy\  {\  static void WINAPI ProxyFun(DL\_REPEAT\_PARAM\_N(param\_count, P, v))\  {\  if (DynFunction::InitFunction())\  DynFunction::GetProxy()(DL\_REPEAT\_N(param\_count, v));\  else\  Policy::template FunctionTrait<DynFunction>::MakeReturn();\  }\  };\  };\  \  template <typename R, DL\_REPEAT\_N(param\_count, typename P), class Policy = CFunProxyValuePolicy<R> >\  struct FUN\_PROXY(param\_count)\  {\  typedef R (WINAPI \*fun\_type)(DL\_REPEAT\_N(param\_count, P));\  typedef R ret\_type;\  template <class DynFunction> struct Proxy:public FUN\_PROXY\_IMPL(param\_count)<R>::template RetProxy<DynFunction, DL\_REPEAT\_N(param\_count, P), Policy>\  {\  };\  }; |

Ключевым в реализации является макрос DECLARE\_FUN\_PROXY(param\_count), который определяет шаблон класса прокси-функции с количеством параметров импортируемой функции, указанным в param\_count. В результате применения этого макроса порождается набор шаблонных классов прокси-функций для количества параметров от 1 до 16. Макросы DL\_REPEAT\_N и DL\_REPEAT\_PARAM\_N формируют список формальных и поименованных параметров соответственно.

В целом, после подстановки макросов, получаемый класс для количества параметров n выглядит так:

|  |
| --- |
| template <typename R, typename P1, typename P2, …, typename Pn , class Policy = CFunProxyValuePolicy<R> >  struct CFunProxyn  {\  typedef R (WINAPI \*fun\_type)(P1, P2, .. , Pn));  typedef R ret\_type;  template <class DynFunction> struct Proxy:public CFunProxynImpln<R>::template RetProxy<DynFunction, P1, P2, .. ,Pn, Policy>  {  };  }; |

Ключевым является вложенный шаблон Proxy, именно он наследует прокси-функцию ProxyFun из CFunProxynImpl. Класс CFunProxynImpl необходим из-за невозможности вернуть тип void при помощи оператора return в Visual C++ 6.0. В качестве обходного маневра используется специализация реализации прокси по типу возвращаемого значения – отдельно для типа void и отдельно для всех остальных типов.

Прокси-функция ProxyFun будет использована в CDynFunction для первоначальной инициализации адреса указателя на функцию:

|  |
| --- |
| static typename proxy\_type::fun\_type &GetProxy()  {  static typename proxy\_type::fun\_type proxy = proxy\_type::template Proxy<type>::ProxyFun;  return proxy;  } |

Для обеспечения возможности реакции на ошибку нахождения функции в модуле используется соответствующая стратегия. Стратегия состоит из класса, вложенного в него шаблона, принимающего в качестве параметра тип ячейки таблицы импортируемых функций и имеющего статическую функцию MakeReturn, которая и вызывается при ошибке поиска адреса функции или при ошибке загрузки библиотеки. На данный момент реализованы 2 стратегии. Одна (CFunProxyThrowPolicy) – выбрасывает исключение (по умолчанию CDynFunException) при ошибке поиска функции\загрузки библиотеки, другая (CFunProxyValuePolicy) – возвращает определенное пользователем значение:

|  |
| --- |
| template <class R>  struct CFunProxyThrowRetTypeTrait  {  template <class F>  struct FunctionTraitImpl  {  static R MakeReturn()  {  F::MakeReturnImpl();  return R();  }  };  };  template <>  struct CFunProxyThrowRetTypeTrait<void>  {  template <class F>  struct FunctionTraitImpl  {  static void MakeReturn()  {  F::MakeReturnImpl();  }  };  };  template<class E = CDynFunException>  struct CFunProxyThrowPolicy  {  template <class DynFunction>  struct FunctionTrait:public CFunProxyThrowRetTypeTrait<typename DynFunction::proxy\_type::ret\_type>::template FunctionTraitImpl<FunctionTrait<DynFunction> >  {  static void MakeReturnImpl()  {  TCHAR szMessage[DynFunction::name\_type::length + 64];  \_stprintf(szMessage, \_T("Can'n resolve procedure <%s>: %d"), DynFunction::name\_type::GetStr(), GetLastError());  throw E(szMessage);  }  };  };  // we need not implement void return type value policy,  // coz void function can only throw on error  template<class R, R value = R()>  struct CFunProxyValuePolicy  {  template <class DynFunction>  struct FunctionTrait  {  static typename DynFunction::proxy\_type::ret\_type MakeReturn()  {  return value;  }  };  }; |

Последние штрихи

Собственно, на этом основные элементы библиотеки реализованы, теперь необходимо описать базовые макросы, которые позволят использовать ее более просто. В библиотеке для объявления импортируемых функций используется интерфейс, сильно напоминающий карту сообщений MFC. Интерфейс состоит из 3-х типов макросов.

Макросы, определяющие модуль и открывающие секцию импортируемых из него функций (DL\_USE\_xxx\_BEGIN);

Макросы, определяющие импортируемые функции (DL\_DECLARE\_FUN\_xxx);

Макрос, закрывающий секцию импорта (DL\_USE\_MODULE\_END).

Таким образом, традиционное объявление динамически импортируемых из библиотеки функций выглядит как

|  |
| --- |
| // объявление библиотеки и пространства имен функций, импортируемых из нее  DL\_USE\_MODULE\_xxx\_BEGIN(name\_space, “some\_lib.dll”)  DL\_DECLARE\_FUN\_xxx(ImportedFunction1Name, … )  DL\_DECLARE\_FUN\_xxx(ImportedFunction2Name, … )  …  DL\_USE\_MODULE\_END() |

Исходя из описанного интерфейса, определены следующие базовые макросы:

Макрос DL\_USE\_MODULE\_LOAD\_POLICY\_BEGIN(nmspace, name, load\_policy)

|  |
| --- |
| #define DL\_USE\_MODULE\_LOAD\_POLICY\_BEGIN(nmspace, name, load\_policy) \  namespace nmspace \  {\  DECLARE\_NAME\_ID(DL\_CAT(\_MODULE\_, nmspace), name)\  typedef delayload::CModule<NAME\_ID(DL\_CAT(\_MODULE\_, nmspace)), load\_policy> module\_type; |

определяет в пространстве имен nmspace (тем самым открывая секцию импорта функций для данной библиотеки) класс модуля, используемого для загрузки библиотеки с именем name, при этом применяя политику загрузки load\_policy. Также в пространстве имен функций импортируемой библиотеки определяется тип module\_type, который представляет собой тип класса модуля для данной библиотеки и может быть использован для управления временем жизни библиотеки, например, для ее выгрузки при помощи статического метода UnloadModule.

Макрос DL\_DECLARE\_FUN\_ERR\_POLICY(name\_id, r, p, pl)

|  |
| --- |
| #define DL\_DECLARE\_FUN\_ERR\_POLICY(name\_id, r, p, pl) \  DECLARE\_NAME\_ID\_A(name\_id, DL\_STRINGIZE(name\_id))\  static r (WINAPI \*&name\_id)(DL\_SEQ\_ENUM(p)) = delayload::CDynFunction<module\_type, NAME\_ID(name\_id), delayload::FUN\_PROXY(DL\_SEQ\_SIZE(p))<r, DL\_SEQ\_ENUM(p), pl > >::GetProxy(); |

определяет ссылку name\_id на указатель на функцию с именем name\_id, типом возвращаемого значения r, списком параметров p и политикой реакции на ошибку загрузки библиотеки\поиска функции pl. Изначально этот указатель указывает на соответствующую прокси-функцию, однако после первого вызова функции указатель указывает непосредственно на саму функцию. Таким образом, использование импортируемой функции из программы тривиально – это обычный вызов функции из пространства имен (nmspace::name\_id).

Неочевидной, но интересной особенностью такой реализации становится то, что автоматически добавляется поддержка UNICODE версий импортируемых функций при подключении заголовков от соответствующих статически линкуемых библиотек, где определены макросы ИмяФункцииW и ИмяФункцииA.

Использование библиотеки

Так как при создании библиотеки одной из основных целей было обеспечение простоты ее использования, то наиболее подходящим интерфейсом объявления импортируемых библиотек и функций оказался интерфейс, внешне напоминающий карты сообщений MFC. В библиотеке определено несколько макросов, которые значительно упрощают ее использование. Это макросы:

DL\_USE\_MODULE\_BEGIN(nmspace, name) – открывает секцию импорта функций из библиотеки. Параметр nmspace – название пространства имен, в которое будет помещены импортируемые функции, name – имя библиотеки, которую необходимо загрузить. Для загрузки используется LoadLibrary;

DL\_USE\_MODULE\_NON\_LOAD\_BEGIN(nmspace, name) – аналогично предыдущему, однако для загрузки используется GetModuleHandle;

DL\_DECLARE\_FUN(name\_id, r, p) – определяет функцию с именем name\_id, типом возвращаемого значения r, и списком типов параметров p в виде (type1)(type2)…(typen). В случае ошибки при загрузке библиотеки\поиске функции из функции возвращается значение r(). Для функций с возвращаемым значением void использование данного макроса не имеет смысла, поскольку распознать ошибку возможным не представится (а в случае Visual C++ 6.0 это просто не скомпилируется);

DL\_DECLARE\_FUN\_ERR(name\_id, r, p, e) – аналогично предыдущему, однако в случае ошибки при загрузке библиотеки\поиске функции возвращается не r(), а значение, указанное в параметре e;

DL\_DECLARE\_FUN\_THROW(name\_id, r, p) – аналогично предыдущему, однако в случае ошибки при загрузке библиотеки\поиске функции выбрасывается исключение CDynFunException;

DL\_USE\_MODULE\_END() – закрывает секцию импорта функций из модуля.

При вызове функции будет использоваться синтаксис nmspace::name\_id.

Рассмотрим пример использования библиотеки в реальной программе:

|  |
| --- |
| #include "stdafx.h"  #include <windows.h>  #include "../delayimphlp.h"  // объявление секции импорта из kernel32.dll  DL\_USE\_MODULE\_BEGIN(kernel, "kernel32.dll")  DL\_DECLARE\_FUN\_ERR(GetProcAddress, FARPROC, (HMODULE)(LPCTSTR), NULL)  DL\_DECLARE\_FUN(GetModuleHandle, HMODULE, (LPCTSTR))  DL\_DECLARE\_FUN\_THROW(InitializeCriticalSection, void, (LPCRITICAL\_SECTION))  DL\_USE\_MODULE\_END()  int main(int argc, char\* argv[])  {  try  {  CRITICAL\_SECTION cs;  HMODULE hm = kernel::GetModuleHandle("ntdll.dll");  kernel::InitializeCriticalSection(&cs);  FARPROC p = kernel::GetProcAddress(hm, "NtQuerySystemInformation");  }  catch (delayload::CDynFunException &E)  {  ::MessageBox(NULL, E.GetMessage(), NULL, MB\_OK | MB\_ICONERROR);  }  return 0;  } |

В данном примере мы загружаем библиотеку kernel32.dll, затем импортируем из нее функции GetProcAddress, GetModuleHandle и InitializeCriticalSection. Как видим, все достаточно просто и тривиально. В случае наличия стандартных заголовков к статически линкуемым библиотекам, где при помощи макросов определены ANSI и UNICODE варианты импортируемых функций, при подключении этих заголовков в зависимости от типа проекта (ANSI или UNICODE), соответствующим образом будут меняться и динамически импортируемые функции, обеспечивая импорт корректных версий функций.

**Заключение**

Итак, в данной статье рассмотрен инструментарий, позволяющий удобно использовать в коде множество динамически загружаемых библиотек и импортируемых из них функций, попутно рассмотрев несколько интересных приемов программирования на C++ в условиях ограниченной поддержки шаблонов. Библиотека получилась, на мой взгляд, достаточно гибкая и хорошо расширяемая, требует достаточно мало ресурсов в плане памяти\кода и получаемый при ее использовании результат в большинстве случаев по быстродействию не уступает статически импортируемым функциям. Многое в ней реализовано так, а не иначе, из расчета поддержки как можно большего количества компиляторов. Библиотека проверялась на работоспособность с Visual C++ 6.0, 7.0 и 7.1, но особых проблем при портировании на другие компиляторы (кроме, пожалуй, линейки от Borland) быть не должно. Автор выражает благодарность всем участникам обсуждения данной библиотеки на форуме RSDN за полезные мысли, советы и поправки. Надеюсь, что данная библиотека поможет хотя бы частично упростить жизнь программистам WinAPI и не только.