**Создание в среде Borland C++ Builder dll, совместимой с Visual C++**

Роман Мананников

Проблемы взаимодействия

Сложность использования dll, созданной с помощью Borland C++ Builder (далее BCB), в проектах, разрабатываемых в средах Microsoft, обусловлена тремя основными проблемами . Во-первых, Borland и Microsoft придерживаются разных соглашений о наименовании (naming convention) функции в dll. В зависимости от того, как объявлена экспортируемая функция, ее имя может быть дополнено компилятором определенными символами. Так, при использовании такого соглашения о вызове (calling convention), как \_\_cdecl, BCB перед именем функции добавляет символ подчеркивания. Visual C++ (далее VC), в свою очередь, при экспорте функции как \_\_stdcall добавит к ее имени помимо подчеркивания также информацию о списке аргументов (символ @ плюс размер списка аргументов в байтах).

|  |
| --- |
| ПРИМЕЧАНИЕ  Использование соглашения \_\_stdcall означает, что вызываемая функция сама удалит из стека свои аргументы. Соглашение \_\_cdecl, наоборот, обязывает очищать стек вызывающую функцию. Объявление функции как \_\_cdecl приведет к некоторому (незначительному) увеличению размера конечного исполняемого файла, поскольку каждый раз после вызова этой функции требуется код по очистке стека, с другой стороны, именно из-за очистки стека вызывающей функцией допускается передача переменного числа параметров. В стек параметры и в том, и в другом случае помещаются справа налево. |

В таблице 1 приведены возможные варианты наименований для экспортируемой функции MyFunction, объявленной следующим образом:

|  |
| --- |
| extern ”C” void \_\_declspec(dllexport) <calling convention> MyFunction(int Param); |

в зависимости от соглашения о вызове (<calling convention>) и компилятора.

|  |  |  |
| --- | --- | --- |
| Соглашение о вызове | VC++ | C++ Builder |
| \_\_stdcall | \_MyFunction@4 | MyFunction |
| \_\_cdecl | MyFunction | \_MyFunction |

Таблица 1. Наименования функций в зависимости от соглашения о вызове и компилятора.

Во-вторых, объектные двоичные файлы (.obj и .lib), создаваемые BCB, несовместимы с объектными файлами VC, и, следовательно, не могут быть прилинкованы к VC-проекту. Это означает, что при желании использовать неявное связывание (linking) c dll необходимо каким-то образом создать .lib-файл (библиотеку импорта) формата, которого придерживается Microsoft.

|  |
| --- |
| ПРИМЕЧАНИЕ  Следует отметить, что до появления 32-разрядной версии Visual C++ 1.0 компиляторы Microsoft использовали спецификацию Intel OMF (Object Module Format – формат объектного модуля). Все последующие компиляторы от Microsoft создают объектные файлы в формате COFF (Common Object File Format – стандартный формат объектного файла). Основной конкурент Microsoft на рынке компиляторов – Borland – решила отказаться от формата объектных файлов COFF и продолжает придерживаться формата OMF Intel. Отсюда и несовместимость двоичных объектных файлов. |

В-третьих, классы и функции-методы классов, экспортируемые из BCB dll, не могут быть использованы в проекте на VC. Причина этого кроется в том, что компиляторы искажают (mangle) имена как обычных функций, так и функций-методов класса (не путайте с разными соглашениями о наименованиях). Искажение вносится для поддержки полиморфизма, то есть для того, чтобы различать функции с одинаковым именем, но разными наборами передаваемых им параметров. Если для обычных функций искажения можно избежать, используя перед определением функции директиву extern ”С” (но при этом, во-первых, на передний план выходит первая проблема – разные соглашения о наименовании функций в dll, а во-вторых, из двух и более функций с одинаковым именем директиву extern ”С” можно использовать только для одной из них, в противном случае возникнут ошибки при компиляции), то для функций-методов класса искажения имени неизбежны. Компиляторы Borland и Microsoft, как вы уже, вероятно, догадались, используют различные схемы внесения искажений. В результате VC-приложения попросту не видят классы и методы классов, экспортируемые библиотеками, скомпилированными в BCB.

|  |
| --- |
| ПРИМЕЧАНИЕ  От редакции: В частности, разновидностями полиморфизма времени компиляции являются перегрузка (ad-hoc полиморфизм) и шаблоны функций (параметрический полиморфизм). |

Эти три проблемы осложняют использование BCB dll из приложений, созданных на VC, но все-таки это возможно. Ниже описаны три способа создания dll совместимой с VC и дальнейшего успешного использования этой dll.

Алгоритмы создания VC-совместимой dll и ее использование

Два из описанных в этом разделе алгоритмов применяют неявное связывание с dll, один – явную загрузку dll. Опишем сначала самый простой способ – использование BCB dll из проекта VC посредством ее явной загрузки в процессе выполнения программы.

Алгоритм с явной загрузкой dll

Применяя данную технику, нам не придется создавать совместимые с VC библиотеки импорта (.lib). Вместо этого добавится ряд действий по загрузке и выгрузке dll в приложении, ее использующем.

Создадим BCB dll (New -> DLL Wizard -> C++ -> Use VCL -> OK), экспортирующую для простоты всего две функции. Одна из функций будет вычислять сумму двух чисел и не будет использовать VCL-классы, а другая будет создавать окно и выводить в VCL-компонент TStringGrid элементы массива, переданного в качестве одного из аргументов.

|  |
| --- |
| ПРИМЕЧАНИЕ  Поскольку действия, производимые функциями, в нашем случае абсолютно не важны, данные примеры не несут смысловой нагрузки, однако стоит обратить внимание на функцию ViewStringGridWnd, которая показывает, что внутри самой dll использовать VCL-классы можно без каких-либо ограничений. |

Листинг 1 - Компилятор Borland C++ Builder 5

ExplicitDll.h

|  |
| --- |
| #ifndef \_EXPLICITDLL\_  #define \_EXPLICITDLL\_  extern "C"  {  int \_\_declspec(dllexport) \_\_cdecl SumFunc(int a, int b);  HWND \_\_declspec(dllexport) \_\_stdcall ViewStringGridWnd(int Count,double\* Values);  }  #endif |

Ключевое слово \_\_declspec с атрибутом dllexport помечает функцию как экспортируемую, имя функции добавляется в таблицу экспорта dll. Таблица экспорта любого PE-файла (.exe или .dll) состоит из трех массивов: массива имен функций (а точнее, массива указателей на строки, содержащие имена функций), массива порядковых номеров функций и массива относительных виртуальных адресов (RVA) функций. Массив имен функций упорядочен в алфавитном порядке, ему соответствует массив порядковых номеров функций. Порядковый номер после некоторых преобразований превращается в индекс элемента из массива относительных виртуальных адресов функций. При экспорте функции по имени имеет место следующая последовательность действий: по известному имени функции определяется ее индекс в массиве имен функций, далее по полученному индексу из массива порядковых номеров определяется порядковый номер функции, затем из порядкового номера, с учетом базового порядкового номера экспорта функций для данного PE-файла, вычисляется индекс, по которому из массива адресов извлекается искомый RVA функции. Помимо экспорта по имени возможен экспорт функций по их порядковым номерам (ordinal). В этом случае последовательность действий для получения индекса элемента из массива относительных виртуальных адресов сводится только к преобразованию порядкового номера функции. Для экспорта функций по номеру используется .def-файл с секцией EXPORTS, где за каждой функцией будет закреплен порядковый номер. При этом в тексте самой dll функции как экспортируемые не помечаются. Подробнее о таблице экспорта можно прочитать в статье по адресу http://www.rsdn.ru/article/baseserv/pe\_coff.xml.



ExplicitDll.cpp

|  |
| --- |
| #include <vcl.h>  #include <grids.hpp>  #include "ExplicitDll.h"  int \_\_cdecl SumFunc(int a, int b)  {  return a + b;  }  HWND \_\_stdcall ViewStringGridWnd(int Count, double\* Values)  {  try  {  // создаем VCL-форму, на которой будет отображен StringGrid,  // и задаем ее основные параметры  TForm\* GridForm = new TForm((TComponent \*)NULL);  GridForm->Caption = "Grid Form";  GridForm->Width = 300;  GridForm->Height = 300;  // создаем компонент StringGrid и устанавливаем его размеры  TStringGrid \*Grid = new TStringGrid(GridForm);  Grid->ColCount = Count + 1;  Grid->RowCount = Count + 1;  // заполняем StringGrid значениями  if (Values != NULL)  for (int i = 0; i < Count; i++)  Grid->Cells[i + 1][i + 1] = Values[i];  // задаем параметры отображения StringGrid в родительском окне  Grid->Parent = GridForm;  Grid->Align = alClient;  // показываем VCL-форму  GridForm->Show();  // возвращаем хэндл VCL-окна клиентскому приложению,  // дабы оно могло это окно при необходимости закрыть  return GridForm->Handle;  }  catch(...)  {  return NULL;  }  }  #pragma argsused  int WINAPI DllEntryPoint(HINSTANCE hinst, unsigned long reason, void\* lpReserved)  {  return 1;  } |

Проанализируем сформированные компилятором наименования экспортируемых функций. Воспользовавшись утилитой impdef.exe, поставляемой совместно с C++Builder (находится в каталоге $(BCB)\Bin, синтаксис командной строки – impdef.exe ExplicitDll.def ExplicitDll.dll), получим следующий .def-файл

ExplicitDll.def

|  |
| --- |
| LIBRARY EXPLICITDLL.DLL  EXPORTS  ViewStringGridWnd @1 ; ViewStringGridWnd  \_SumFunc @2 ; \_SumFunc  \_\_\_CPPdebugHook @3 ; \_\_\_CPPdebugHook |

Поскольку в данном примере экспортируемая функция ViewStringGridWnd использует соглашение \_\_stdcall, ее имя осталось неизменным (см. таблицу 1), следовательно, для вызова этой функции VC-приложение воспользуется именем ViewStringGridWnd (например, при вызове GetProcAddress), а вот для вызова функции SumFunc использовать придется имя \_SumFunc. Очевидно, что осуществлять вызов функции, пользуясь ее измененным именем, неудобно само по себе, а тем более, если dll пишет один программист, а работает с ней другой. Для того чтобы при использовании \_\_cdecl-соглашения экспортируемые функции можно было использовать с их истинными именами (без символов подчеркивания), необходимо об этом позаботиться заранее, то есть на этапе создания самой dll. Для этого создается .def-файл (это можно сделать в любом текстовом редакторе), в котором определяется секция EXPORTS, содержащая псевдоним (alias) для каждой экспортируемой \_\_cdecl-функции. В нашем случае он будет выглядеть следующим образом

ExplicitDllAlias.def

|  |
| --- |
| EXPORTS  ; VC funcname = BCB funcname  SumFunc = \_SumFunc |

То есть, у функции, экспортируемой как \_SumFunc, будет псевдоним SumFunc, который мы исключительно для удобства делаем идентичным оригинальному имени этой функции в коде (хотя псевдоним может быть каким угодно).

Созданный .def-файл добавляется (Project -> Add to Project) к проекту dll. После компиляции, проанализировав dll c помощью impdef.exe, получим следующее

ExplicitDll.def

|  |
| --- |
| libRARY EXPLICITDLL.DLL  EXPORTS  SumFunc @4 ; SumFunc  ViewStringGridWnd @2 ; ViewStringGridWnd  \_SumFunc @1 ; \_SumFunc  \_\_\_CPPdebugHook @3 ; \_\_\_CPPdebugHook |

Имеем на одну экспортируемую функцию больше, но при этом реальное количество функций в dll осталось неизменным, а функция с именем SumFunc (функция-псевдоним) является ссылкой на свой оригинал, то есть на функцию, экспортируемую под именем \_SumFunc.

|  |
| --- |
| ПРИМЕЧАНИЕ  Более правильным будет сказать, что функция-псевдоним попросту добавляется в таблицу экспорта dll: ее имя SumFunc добавляется в массив имен функций, а в массив порядковых номеров добавляется присвоенный ей порядковый номер. Однако соответствующий функции-псевдониму RVA в массиве относительных виртуальных адресов будет равен RVA функции с именем \_SumFunc. Убедиться в этом можно последовательно вызывая GetProcAddress для имен функций SumFunc и \_SumFunc и анализируя возвращаемый адрес (можно, разумеется, воспользоваться различными программами, позволяющими просмотреть содержимое исполняемого файла). В обоих случаях адрес функции будет одинаков. |

Таким образом, с помощью .def-файла псевдонимов при экспорте функций, определенных как \_\_cdecl, мы избавляем пользователей от необходимости вызова функций по их измененным именам, хотя и такая возможность остается.

|  |
| --- |
| ПРЕДУПРЕЖДЕНИЕ  Поскольку \_\_stdcall- и \_\_cdecl-функции по-разному работают со стеком, не пытайтесь из клиентского приложения вызывать \_\_stdcall-функции как \_\_cdecl, и наоборот, иначе стек будет поврежден, и дальнейшее выполнение приложения будет невозможно. |

В результате изложенного выше мы получили dll, экспортирующую функции с именами SumFunc и ViewStringGridWnd. При этом их названия не зависят от того, какое соглашение о вызове использовалось при объявлении этих функций. Теперь рассмотрим пример использования нашей dll в приложении VC. Создадим в среде Visual C++ 6.0 (или Visual C++ 7.0) простое MFC-приложение, которое будет представлять собой обычное диалоговое окно (File -> New -> MFC AppWizard(exe) -> Dialog based -> Finish). Добавим к исходному диалогу две кнопки: кнопку “SumFunc” и кнопку “ViewStringGridWnd”. Затем для каждой кнопки создадим обработчик события BN\_CLICKED: OnSumFunc() и OnViewStringGridWnd() соответственно. Нам также понадобятся обработчики сообщений для событий формы WM\_CREATE и WM\_DESTROY. Полный рабочий код этого приложения находится в примерах к статье, здесь же будет приведена только часть, демонстрирующая работу с нашей dll, поскольку оставшаяся часть кода генерируется средой разработки.

Листинг 2 - Компилятор Visual C++ 6.0

UsingExplicitDLLDlg.cpp

|  |
| --- |
| // код, генерируемый средой разработки  …  // хэндл тестируемой DLL  HINSTANCE hDll = NULL;  // тип указателя на функцию ViewStringGridWnd  typedef HWND (\_\_stdcall \*ViewStringGridWndProcAddr) (int Count, double\* Values);  // хэндл окна с VCL-компонентом StringGrid  HWND hGrid = NULL;  // тип указателя на функцию SumFunc  typedef int (\_\_cdecl \*SumFuncProcAddr) (int a, int b);  // код, генерируемый средой разработки  …  // обработчик нажатия кнопки SumFunc  void CUsingExplicitDLLDlg::OnSumFunc()  {  // указатель на функцию SumFunc  SumFuncProcAddr ProcAddr = NULL;  if( hDll != NULL )  {  // получение адреса функции  ProcAddr = (SumFuncProcAddr) GetProcAddress(hDll, "SumFunc");  if( ProcAddr != NULL )  {  // вызов функции  int result = (ProcAddr)(5, 6);  // отображение результата в заголовке диалога  char str[10];  this->SetWindowText(itoa(result, str ,10));  }  }  }  // обработчик нажатия кнопки ViewStringGridWnd  void CUsingExplicitDLLDlg::OnViewStringGridWnd()  {  // указатель на функцию ViewStringGridWnd  ViewStringGridWndProcAddr ProcAddr = NULL;  if( hDll != NULL )  {  // получение адреса функции  ProcAddr = (ViewStringGridWndProcAddr) GetProcAddress(hDll,  "ViewStringGridWnd");  if( ProcAddr != NULL )  {  // инициализация аргументов  const int count = 5;  double Values[count] = {2.14, 3.56, 6.8, 8, 5.6564};  // закрываем ранее созданное окно, чтобы они не плодились  if( hGrid != NULL )  ::SendMessage(hGrid, WM\_CLOSE, 0, 0);  // вызов функции  hGrid = (ProcAddr)(count, Values);  }  }  }  // обработчик события окна WM\_DESTROY  void CUsingExplicitDLLDlg::OnDestroy()  {  CDialog::OnDestroy();    // закрываем окно с компонентом StringGrid, если оно было создано  if( hGrid != NULL )  ::SendMessage(hGrid, WM\_CLOSE, 0, 0);  // выгрузка dll из памяти  FreeLibrary( hDll );  }  // обработчик события окна WM\_CREATE  int CUsingExplicitDLLDlg::OnCreate(LPCREATESTRUCT lpCreateStruct)  {  if (CDialog::OnCreate(lpCreateStruct) == -1)  return -1;    // загрузка dll в память  hDll = LoadLibrary("ExplicitDll.dll");    return 0;  } |

Явная загрузка dll имеет как преимущества, так и недостатки. В нашем случае большим плюсом является то, что явная загрузка избавляет от какого бы то ни было взаимодействия с исходным кодом dll, в частности нет необходимости подключать заголовочный .h-файл с объявлениями функций. Клиентское приложение компилируется и работает независимо от используемой dll, а случаи неудачной загрузки библиотеки или неудачного получения адреса функции всегда можно обыграть так, чтобы они не повлияли на дальнейшее выполнение основного приложения.

|  |
| --- |
| ПРИМЕЧАНИЕ  Следует отметить, что использование экспортируемых unmanaged-функций из управляемого кода (managed code) в .NET осуществляется исключительно посредством явной загрузки dll. К процессу вызова функции в этом случае помимо стандартных шагов (таких как загрузка dll в память посредством LoadLibrary, получение адреса требуемой функции с помощью GetProcAddress и непосредственно вызов), добавляется также процесс маршалинга (marshaling), то есть процесс преобразования типов данных .NET в их аналоги в традиционном двоичном коде (при проталкивании аргументов в стек) и обратно (при анализе возвращаемого значения). Для указания, что метод импортируется из dll, используется атрибут DllImport, параметры которого содержат информацию, необходимую для вызова LoadLibrary и GetProcAddress. |

Таким образом, для вызова экспортируемой функции из dll, скомпилированной в BCB, необходимо выполнить следующую последовательность действийя:

Объявить экспортируемые функции либо как \_\_cdecl, либо как \_\_stdcall. Если используется только соглашение \_\_stdcall, пропускаем пункт 3.

Поместить объявления функций в блок extern ”С”. Не экспортировать классы и функции-члены классов, поскольку это все равно не удастся.

Если экспортируются функции с соглашением о вызове \_\_cdecl, то добавить к проекту .def-файл с псевдонимами для каждой такой функции.

Откомпилировать dll.

Создать клиентский (то есть использующий BCB библиотеку) VC-проект.

Скопировать созданную BCB dll в папку с клиентским VC-приложением.

Загрузить dll из клиентского приложения в память при помощи LoadLibrary.

Получить адрес требуемой функции с помощью GetProcAddress и присвоить его указателю на функцию.

Вызвать функцию с помощью указателя на нее.

По окончании использования выгрузить dll из памяти с помощью FreeLibrary.

Алгоритм с неявным связыванием для экспорта (импорта) \_\_cdecl-функций

Как следует из названия раздела, данный способ предназначен для экспорта (а на клиентской стороне – для импорта) функций с \_\_cdecl-соглашением о вызове. Чтобы воспользоваться неявным связыванием, прежде всего, необходимо создать объектный .lib-файл (библиотеку импорта), содержащий ссылку на dll и перечень находящихся в dll функций. Данный объектный файл можно создать по .def-файлу экспорта библиотеки с помощью утилиты lib.exe. При этом полученный .lib-файл будет в нужном нам формате COFF, поскольку компилятор VC придерживается именно этой спецификации (утилита lib.exe поставляется совместно с VC и умеет создавать библиотеки импорта только по .def-файлу). Готовый .lib-файл прилинковывается к клиентскому проекту.

При неявном связывании приложение не подозревает, что использует dll, поэтому функции, вызываемые из динамической библиотеки, как и любые другие, должны быть объявлены в тексте клиентской программы. Для объявления функций воспользуемся исходным заголовочным файлом BCB dll, но функции в нем должны быть помечены уже не как \_\_declspec(dllexport), а как \_\_declspec(dllimport), то есть как импортируемые извне, поскольку по отношению к клиентскому приложению эти функции являются именно импортируемыми.

Исходный текст dll на этот раз будет выглядеть следующим образом:

Листинг 3 - Компилятор Borland C++ Builder 5

ImplicitLinking\_cdecl.h

|  |
| --- |
| #ifndef \_IMPLICITDLL\_  #define \_IMPLICITDLL\_  // если макрос-идентификатор \_DLLEXPORT\_ был определен ранее,  // то макрос \_DECLARATOR\_ пометит функцию как экспортируемую,  // в противном случае функция будет помечена как импортируемая.  // Данная конструкция из директив препроцессора позволяет  // воспользоваться заголовочным файлом библиотеки как на этапе  // создания DLL, так и на этапе ее использования, а именно, при  // неявном связывании.  #ifdef \_DLLEXPORT\_  #define \_DECLARATOR\_ \_\_declspec(dllexport)  #else  #define \_DECLARATOR\_ \_\_declspec(dllimport)  #endif  extern "C"  {  int \_DECLARATOR\_ \_\_cdecl SumFunc(int a, int b);  HWND \_DECLARATOR\_ \_\_cdecl ViewStringGridWnd(int Count, double\* Values);  }  #endif |

ImplicitLinking\_cdecl.cpp

|  |
| --- |
| #include <vcl.h>  #include <grids.hpp>  // определение \_DLLEXPORT\_, дабы вместо макроса \_DECLARATOR\_  // в заголовочном файле было подставлено \_\_declspec(dllexport),  // и функции были объявлены как экспортируемые  #define \_DLLEXPORT\_  #include "ImplicitLinking\_cdecl.h"  int \_\_cdecl SumFunc( int a, int b )  { // тело функции такое же как в предыдущем разделе  }  HWND \_\_cdecl ViewStringGridWnd( int Count, double\* Values )  { // тело функции такое же как в предыдущем разделе  }  #pragma argsused  int WINAPI DllEntryPoint(HINSTANCE hinst,  unsigned long reason,  void\* lpReserved)  {  return 1;  } |

Основная возникающая при этом проблема заключается в том, что, согласно таблице 1, функции с \_\_cdecl-соглашением о вызове будут экспортироваться с символом подчеркивания, следовательно, .lib-файл, созданный по .def-файлу экспорта библиотеки, будет содержать измененные имена функций. С другой стороны, во-первых, компилятор VC будет ожидать неизмененных наименований \_\_cdecl-функций, потому что сам VC, экспортируя функции с \_\_cdecl-соглашением о вызове, ничего к их наименованию не добавляет, а во-вторых, заголовочный файл BCB dll, подключаемый к клиентскому приложению, содержит объявления функций с их реальными (без символа подчеркивания) именами. В результате этого, если в тексте клиентского приложения встретится хотя бы один вызов нашей функции, то VC при связывании попытается найти описание этой импортируемой функции в добавленной к проекту библиотеке импорта (.lib-файле), с тем, чтобы добавить соответствующую запись в таблицу импорта приложения. Но из-за несоответствия имен функций в заголовочном и объектном файлах линковщик, естественно, в .lib-файле ничего не найдет, о чем не замедлит выдать сообщение (например, такое - error LNK2001: unresolved external symbol \_\_imp\_\_SumFunc).

|  |
| --- |
| ПРИМЕЧАНИЕ  Таблица импорта любого PE-файла содержит массив структур IMAGE\_IMPORT\_DESCRIPTOR. Каждая такая структура соответствует одной из dll, с которой неявно связан PE-файл. Структура IMAGE\_IMPORT\_DESCRIPTOR среди прочих полей содержит поле с RVA строки-наименования dll, которой она соответствует, и два поля с RVA массивов двойных слов, предназначенных для хранения информации об импортируемых функциях. При запуске приложения загрузчик PE-файлов заполняет один из этих массивов (так называемую таблицу адресов импорта) адресами импортируемых функций, загрузив перед этим dll, в которой эти функции находятся. Адрес импортируемой функции вычисляется как сумма адреса, по которому была загружена экспортирующая данную функцию dll, и смещения (RVA) самой функции относительно начала dll. |

Описанную выше проблему несоответствия заголовочного и объектного файлов можно решить двумя способами – снова воспользоваться рассмотренным в предыдущем разделе .def-файлом с псевдонимами или использовать в заголовочном файле нашей библиотеки директиву препроцессора #define.

Использование псевдонимов

Следуя этому способу, создаем и добавляем к проекту BCB dll следующий .def-файл:

ImplicitLinkingAliases.def



|  |
| --- |
| EXPORTS  ; MSVC name = Borland name  SumFunc = \_SumFunc  ViewStringGridWnd = \_ViewStringGridWnd |

После компиляции наша dll будет экспортировать функции

ImplicitLinking\_cdecl.def

|  |
| --- |
| libRARY IMPLICITLINKING\_CDECL.DLL  EXPORTS  SumFunc @4 ; SumFunc  ViewStringGridWnd @5 ; ViewStringGridWnd  \_SumFunc @1 ; \_SumFunc  \_ViewStringGridWnd @2 ; \_ViewStringGridWnd  \_\_\_CPPdebugHook @3 ; \_\_\_CPPdebugHook |

Таким образом, в таблицу экспорта dll добавляются функции-псевдонимы, имена которых соответствуют функциям, объявленным в заголовочном файле нашей библиотеки. Для полного соответствия (хотя этого можно и не делать) удалим из ImplicitLinking\_cdecl.def упоминания обо всех посторонних для приложения-клиента функциях, так как заголовочный файл содержит объявления только двух функций. В результате получим .def-файл готовый для генерации из него объектного .lib-файла:

ImplicitLinking\_cdecl.def

|  |
| --- |
| libRARY IMPLICITLINKING\_CDECL.DLL  EXPORTS  SumFunc @4 ; SumFunc  ViewStringGridWnd @5 ; ViewStringGridWnd |
| ПРИМЕЧАНИЕ  В единственной статье, которую мне удалось найти по данной теме (на сайте bcbdev.com), рекомендовалось, помимо удаления из .def-файла посторонних функций, заменить наименование секции EXPORTS на IMPORTS. Делать этого не следует по той простой причине, что утилита lib.exe (по крайней мере, поставляемая с 6-ой и 7-ой Visual Studio) секцию IMPORTS не поддерживает, поэтому игнорирует все последующие описания функций и создает пустой .lib-файл. Утилита lib.exe находится в каталоге $(VC)\Bin, но запустить ее обычно с первого раза не удается, поскольку для работы ей требуется библиотека mspdb60.dll (для lib.exe, поставляемой с Visual Studio 7 – mspdb70.dll). mspdb60.dll лежит в папке $(Microsoft Visual Studio)\Common\MSDev98\Bin, а mspdb70.dll – в папке $(Microsoft Visual Studio .NET)\Common7\IDE. | | |

С помощью утилиты lib.exe создадим необходимый для неявного связывания .lib-файл в формате COFF, для этого в командной строке наберем

|  |
| --- |
| lib.exe /def:ImplicitLinking\_cdecl.def |

либо

|  |
| --- |
| lib.exe /def:ImplicitLinking\_cdecl.def /out:ImplicitLinking\_cdecl.lib |

Полученный .lib-файл добавим к проекту VC-клиента (Project -> Add To Project -> Files…).

Использование директивы препроцессора #define

Теперь рассмотрим способ, позволяющий добиться одинаковых названий функций в заголовочном и объектном (.lib) файлах с помощью директивы #define. Перепишем заголовочный файл нашей BCB-библиотеки следующим образом

Листинг 4 - Компилятор Borland C++ Builder 5

ImplicitLinking\_cdecl.h

|  |
| --- |
| #ifndef \_IMPLICITDLL\_  #define \_IMPLICITDLL\_  #ifdef \_DLLEXPORT\_  #define \_DECLARATOR\_ \_\_declspec(dllexport)  #else  #define \_DECLARATOR\_ \_\_declspec(dllimport)  #endif  extern "C"  {  // при компиляции в VC к оригинальным наименованиям  // функций добавятся символы подчеркивания, таким образом  // имена объявляемых функций совпадут с их именами в таблице  // экспорта DLL и, следовательно, .lib-файле  #ifdef \_MSC\_VER  #define SumFunc \_SumFunc  #define ViewStringGridWnd \_ViewStringGridWnd  #endif  int \_DECLARATOR\_ \_\_cdecl SumFunc(int a, int b);  HWND \_DECLARATOR\_ \_\_cdecl ViewStringGridWnd(int Count, double\* Values);  }  #endif |

При компиляции клиентского VC-приложения в подключенном к проекту заголовочном файле dll (ImplicitLinking\_cdecl.h) к наименованию каждой функции с помощью директив #define добавляется символ подчеркивания (макрос \_MSC\_VER определяется компилятором VC по умолчанию). Поскольку из BCB dll \_\_cdecl-функции экспортируются таким же образом, то есть с добавлением символа подчеркивания, то устанавливается соответствие имен экспортируемых и объявленных функций. Макросы #define распространяют свое влияние и на весь последующий код приложения, что позволяет в тексте программы при вызове импортируемой функции пользоваться ее оригинальным именем, которое при компиляции будет дополнено необходимым магическим символом подчеркивания. Таким образом, мы идем на поводу у фирмы Borland и в клиентском приложении завуалированно используем для вызова функций из нашей dll имена, измененные компилятором BCB. Именно необходимость использования измененных имен (пусть и не в открытую благодаря define-трюку), на мой взгляд, является существенным недостатком этого способа, так как, например, при желании явно (см. раздел “Алгоритм с явной загрузкой dll”) использовать dll придется оперировать измененными именами функций. Не развивая дальше эту тему, скажу, что если BCB dll создается с четким намерением использовать ее в VC-приложениях, то лучше добавлять к проекту библиотеки .def-файл с удобными для пользователей именами-псевдонимами функций.

К достоинствам данного способа (define-трюка) можно отнести его простоту и, как бы это ни противоречило сказанному в предыдущем абзаце, отсутствие необходимости добавлять к таблице экспорта dll псевдонимы функций. Несмотря на все удобства использования псевдонимов, таблица экспорта (а следовательно, и сама dll) при этом увеличивается в размерах. Да и создание .def-файла псевдонимов при большом количестве функций не добавляет приятных эмоций.

После компиляции dll с помощью impdef.exe получаем .def-файл экспорта, из которого утилитой lib.exe создаем объектный .lib-файл и добавляем его к клиентскому VC-проекту.

Листинг клиентского приложения, код которого в данном случае не зависит от способа решения проблемы несоответствия наименований функций в заголовочном и объектном файлах библиотеки, представлен ниже. Как и в предыдущем разделе, это диалоговое окно с двумя кнопками. Интересующий нас код сосредоточен в обработчиках событий нажатия кнопок диалога.

Листинг 5 - Компилятор Visual C++ 6.0

UsingImplicitLinking\_cdeclDlg.cpp

|  |
| --- |
| // код, генерируемый средой разработки  …  // хэндл окна с VCL-компонентом StringGrid  HWND hGrid = NULL;  // подключаем заголовочный файл библиотеки  #include "ImplicitLinking\_cdecl.h"  // код, генерируемый средой разработки  …  void CUsingImplicitLinkng\_cdeclDlg::OnSumFunc()  {  // вызываем функцию SumFunc из dll  int res = SumFunc(5, 9);  // выводим результат в заголовок диалогового окна  char str[10];  this->SetWindowText(itoa(res, str ,10));  }  void CUsingImplicitLinkng\_cdeclDlg::OnViewStringGridWnd()  {  // инициализация аргументов  const int count = 5;  double Values[count] = {2.14, 3.56, 6.8, 8, 5.6564};  // закрываем ранее созданное окно, чтобы они не «плодились»  if( hGrid != NULL )  ::SendMessage(hGrid, WM\_CLOSE, 0, 0);  // вызываем функцию ViewStringGridWnd из dll  hGrid = ViewStringGridWnd(count, Values);  }  void CUsingImplicitLinkng\_cdeclDlg::OnDestroy()  {  CDialog::OnDestroy();    // закрываем окно с компонентом StringGrid, если оно было создано  if( hGrid != NULL )  ::SendMessage(hGrid, WM\_CLOSE, 0,0);  } |

Основным преимуществом неявной загрузки dll является именно неявность использования dll со стороны клиентского приложения. Другими словами, приложение, вызывая функции, не подозревает, что они могут находиться где-то во внешнем модуле. Результатом является упрощение кода программы. К недостаткам следует отнести тот факт, что dll находится в памяти в течение всей работы программы, неявно ее использующей. Загрузка dll осуществляется при загрузке приложения – загрузчик PE-файлов, просматривая каждую запись в таблице импорта приложения, загружает соответствующую этой записи dll. Следовательно, если используемых библиотек много, загрузка основной программы может затянуться. В случае отсутствия неявно используемой dll приложение вообще не запустится.

Итоговый алгоритм с неявным связыванием для экспорта (импорта) \_\_cdecl-функций состоит из следующей последовательности действий (см. также Демонстрационный проект):

1. Объявить экспортируемые функции как \_\_cdecl.

2. Поместить объявления функций в блок extern ”С”, при этом не экспортировать классы и функции-члены классов.

3. В заголовочный файл для возможности его дальнейшего использования на клиентской стороне вставить:

|  |
| --- |
| #ifdef \_DLLEXPORT\_  #define \_DECLARATOR\_ \_\_declspec(dllexport)  #else  #define \_DECLARATOR\_ \_\_declspec(dllimport)  #endif |

и добавить макрос \_DECLARATOR\_ к объявлению каждой функции, например,

|  |
| --- |
| int \_DECLARATOR\_ \_\_cdecl SumFunc( int a, int b ); |

4. Далее либо создать и добавить к проекту .def-файл с псевдонимами для каждой функции, либо добавить в заголовочный файл библиотеки следующее:

|  |
| --- |
| #ifdef \_MSC\_VER  #define FuncName1 \_FuncName1  #define FuncName2 \_FuncName2  #define FuncNameN \_FuncNameN  #endif |

Если использовался #define-трюк, то пункт 7 нужно будет пропустить.

5. Скомпилировать BCB dll.

6. С помощью impdef.exe создать .def-файл с наименованиями экспортируемых функций.

7. Если в пункте 4 воспользовались псевдонимами, удалить из .def-файла экспорта неиспользуемые наименования функций, оставив только псевдонимы.

8. Создать клиентский VC-проект.

9. Из .def-файла экспорта библиотеки при помощи утилиты lib.exe создать объектный .lib-файл формата COFF и добавить его к клиентскому VC-приложению.

10. Скопировать BCB dll и ее заголовочный файл в папку с клиентским VC-проектом.

11. В клиентском приложении подключить заголовочный файл dll.

12. Вызвать в теле программы необходимые функции, не задумываясь над тем, что они расположены во внешней dll.

Алгоритм с неявным связыванием для экспорта (импорта) \_\_stdcall-функций

Как уже упоминалось выше, утилита lib.exe может создавать библиотеку импорта только из .def-файла экспорта, при чем lib.exe при этом никак не взаимодействует с самой dll. Однако .def-файл не содержит никакой информации, касаемой соглашений о вызове, которых придерживаются экспортируемые функции. Следовательно, и lib.exe, работая исключительно с .def-файлом, не сможет уловить, что имеет дело с \_\_stdcall-функциями, и, как результат, не сможет в .lib-файле отобразить функции согласно Microsoft-соглашению о наименовании для \_\_stdcall-функций. Таким образом, учитывая из предыдущего раздела, что для \_\_cdecl-функций lib.exe генерирует вполне работоспособный .lib-файл, приходим к следующему выводу: утилита lib.exe не способна генерировать библиотеки импорта для dll, экспортирующих \_\_stdcall-функции. Людям, пожелавшим или вынужденным (а после прочтения этого раздела думаю только вынужденным) использовать BCB dll с \_\_stdcall-функциями в VC, этот раздел посвящается.

Исходный код BCB dll остался таким же, как в предыдущем разделе (см. Листинг 3), только ключевое слово \_\_cdecl везде необходимо заменить ключевым словом \_\_stdcall.

Известно, что при создании VC dll вместе с ней среда генерирует .lib-файл (библиотеку импорта), который представлен, естественно, в нужном нам формате COFF, и в котором корректно будут отображаться \_\_stdcall-функции. Поэтому создадим (File -> New… -> Win32 Dynamic-Link Library -> OK -> An empty DLL project -> Finish) ложную (dummy) VC dll, которая будет экспортировать тот же набор функций, что и BCB dll. Реализация функций в ложной dll абсолютно не важна, важны исключительно их наименования. Помимо одинаковых наименований экспортируемых функций у ложной и исходной библиотек должны совпадать имена, поскольку .lib-файлы содержат наименования dll. Можно воспользоваться исходными текстами BCBdll, скопировав .h- и .cpp-файлы в директорию к ложной dll, затем добавив их к проекту (Project -> Add To Project -> Files…) и удалив тела всех функций. Если функция возвращает значение, то оставляем оператор return и возвращаем в соответствии с типом все, что угодно (можно 0, NULL и т.д.). Поскольку тела функций будут пустыми, большую часть директив #include с подключаемыми заголовочными файлами также можно удалить. В итоге получим согласно нашему примеру следующий код ложной dll:

Листинг 6 - Компилятор Visual C++ 6.0

ImplicitLinking\_stdcallDummy.h

|  |
| --- |
| #ifdef \_DLLEXPORT\_  #define \_DECLARATOR\_ \_\_declspec(dllexport)  #else  #define \_DECLARATOR\_ \_\_declspec(dllimport)  #endif  extern "C"  {  int \_DECLARATOR\_ \_\_stdcall SumFunc(int a, int b);  HWND \_DECLARATOR\_ \_\_stdcall ViewStringGridWnd(int Count, double\* Values);  } |

ImplicitLinking\_stdcallDummy.cpp

|  |
| --- |
| #define \_DLLEXPORT\_  #include <windows.h>  #include "ImplicitLinking\_stdcallDummy.h"  int \_\_stdcall SumFunc(int a, int b)  {  return 0;  }  HWND \_\_stdcall ViewStringGridWnd(int Count, double\* Values)  {  return NULL;  } |

Согласно таблице 1, VC экспортирует \_\_stdcall-функции, добавляя к их наименованию информацию о списке аргументов и символ подчеркивания. Следовательно, в объектном .lib-файле будут имена, отличные от оригинальных имен функций, объявленных в заголовочном файле, и тем более отличные от наименований функций, экспортируемых из BCB dll, так как \_\_stdcall-функции компилятор BCB экспортирует без изменений. Избавляться от этого несоответствия будем снова посредством .def-файла. Для нашего примера он будет следующим:

DummyDef.def

|  |
| --- |
| libRARY ImplicitLinking\_stdcall.dll  EXPORTS  SumFunc  ViewStringGridWnd |

Строка с именем библиотеки (LIBRARY) в .def-файле не обязательна, но если она есть, то имя, указанное в ней, в точности должно совпадать с именами ложной и исходной dll. Добавляем .def-файл к VC-проекту, перекомпилируем и получаем ложную dll и необходимую нам библиотеку импорта, содержащую корректное описание экспортируемых \_\_stdcall-функций. .lib-файл, доставшийся в наследство от ложной dll, должен добавляться (прилинковываться) к любому VC-проекту, который собирается использовать нашу исходную BCB dll.

Пример VC-приложения, импортирующего \_\_stdcall-функции, такой же, как и в предыдущем разделе (см. Листинг 5). Не забудьте в примере подключить (#include) нужный заголовочный файл BCB dll и добавить к проекту нужную библиотеку импорта.

Алгоритм с неявным связыванием для экспорта (импорта) \_\_stdcall-функций (см. также Демонстрационный проект, ImplicitLinkingDll\_stdcall.zip):

Объявить экспортируемые функции как \_\_stdcall.

Поместить объявления функций в блок extern ”С”. Не экспортировать классы и функции-члены классов.

Скомпилировать BCB dll.

Поскольку создать корректную библиотеку импорта с помощью утилиты lib.exe не удается, создать ложную VC dll, которая содержит такой же набор функций, как и исходная BCB dll.

Проверить идентичность названий ложной dll и dll исходной, названия должны совпасть.

Если для ложной библиотеки используются исходные тексты BCB dll, то удалить тела функций, если не используются, то создать пустые функции с такими же именами и сигнатурами, как в исходной dll.

Дабы предотвратить изменение имен функций при экспорте, добавить к VC-проекту ложной библиотеки .def-файл с секцией EXPORTS, в которой просто перечислены оригинальные наименования всех экспортируемых функций.

Скомпилировать ложную dll и получить необходимый .lib-файл с корректным отображением \_\_stdcall-функций.

Создать клиентский VC-проект и добавить к нему полученный .lib-файл.

Скопировать BCB dll и ее заголовочный файл в папку с клиентским VC-проектом.

В клиентском приложении подключить заголовочный файл.

Вызвать в тексте программы необходимые функции, не задумываясь над тем, что они расположены во внешней dll.

**Заключение**

Как вы могли убедиться, обеспечение успешного взаимодействия BCB dll и клиентского VC-приложения является нетривиальной задачей. Однако такое взаимодействие становится необходимым в случаях, когда использование VCL и C++ Builder-а при разработке отдельных частей приложения является более предпочтительным (например, в силу временных затрат). Используя описанные в статье алгоритмы, вы сможете создавать и успешно использовать BCB dll из VC-проекта.