А.В. Леденёв, И.А. Семёнов, В.А. Сторожевых

Динамически загружаемые библиотеки: структура, архитектура и применение (часть 3)

DLL — сокращение от Dynamic Link Library (динамически загружаемая библиотека). С формальной точки зрения DLL — особым образом оформленный относительно независимый блок исполняемого кода. DLL используют множество приложений. Все приложения для ОС Windows так или иначе используют динамические библиотеки.

Данный материал является завершением работы, посвященной особенностям реализации DLL в различных средах и для различных целей, опубликованной в № 2 и 4 журнала за этот год.

Проблемы разработки и использования DLL в различных средах

Как было сказано ранее, DLL — это программные компоненты, которые оформлены специальным образом и позволяют добиться интеграции программного кода, написанного на различных языках программирования. Сказанное следует понимать в том смысле, что заявленная среда разработки в общем случае не обязана совпадать со средой, в которой данная DLL используется. Более того, клиент может даже не подозревать, что DLL написана на другом языке, нежели тот, который используется при ее вызове. В ряде случаев следует помнить о небольших подводных камнях, которые могут появиться на пути ничего не подозревающего разработчика.

Проблема 1. Декорирование имен

Данная проблема подробно обсуждалась в разделе «Декорирование имен». Там же рассмотрены пути ее решения. Если при разработке DLL забыть об этом аспекте, то в приложении, использующем неявную загрузку DLL, линкер может выдать ряд подозрительных ошибок о невозможности разрешения того или иного имени функции.

Проблема 2. Правила занятия и освобождения стека

Каждый из языков применяет различные стратегии для занятия и освобождения стека. Аргументы могут помещаться в стек по очереди слева-направо или справа-налево, стек может освобождаться вызывающим или вызываемым кодом.

Приведем варианты псевдокода для каждого возможного случая (табл. 1).

Таблица 1

Различные варианты вызова функции

Варианты: порядок размещения аргументов/ тип освобождения стека	
Вариант 1: слева-направо/ вызывающим кодом	**Вариант 2: справа-налево/ вызывающим кодом**

`; размещение операндов в стеке`
`push arg1`
`push arg2`
`...`
`push argN`
`call Func`
`; очистка стека`
`pop argN`
`...`
`pop arg2`
`pop arg1`
`----------`
`Func:`
`  ...`
`  ret`
	`; размещение операндов в стеке`
	`push argN`
	`...`
	`push arg2`
	`push arg1`
	`call Func`
	`; очистка стека`
	`pop arg1`
	`pop arg2`
	`...`
	`pop argN`
	`----------`
	`Func:`
	`  ...`
	`  ret`

Окончание табл. 1

Варианты: порядок размещения аргументов/ тип освобождения стека	
Вариант 3: слева-направо/ вызываемым кодом	**Вариант 4: справа-налево/ вызываемым кодом**

`; размещение операндов`
`; в стеке`
`push arg1`
`push arg2`
`...`
`push argN`
`call Func`
`...`
`---------- `
`Func:`
`  ...`
`  ; очистка стека`
`  pop argN`
`  ...`
`  pop arg2`
`  pop arg1`
`  ; возврат`
`  ret`
	`; размещение операндов`
	`; в стеке`
	`push argN`
	`...`
	`push arg2`
	`push arg1`
	`call Func`
	`...`
	`----------`
	`Func:`
	`  ...`
	`  ; очистка стека`
	`  pop arg1`
	`  pop arg2`
	`  ...`
	`  pop argN`
	`  ; возврат`
	`  ret`

В табл. 1 при помощи псевдокода показаны различные варианты вызова функции. Оператор **push** описывает команду сохранения операнда в стеке, оператор **pop** — его извлечения. Команда **call** осуществляет вызов функции, а команда **ret** — возврат из нее.

Пунктирная линия предполагает, что данная функция (изначально) может располагаться за пределами вызывающего ее кода.

Замечание.

Реальные компиляторы генерируют примерно следующее для кода с очисткой стека вызывающей функцией:

```
push   arg1
push   arg2
push   arg3
call   Func
add    esp, 0x0C      ; либо sub esp, 0xFFFFFFF4 -
                      что то же самое
```

Для кода с очисткой стека вызываемой функцией генерируется такой код:

```
push      arg1
push      arg2
push      arg3
call      Func

Func...
...
ret       0x0C
```

Таким образом, даже при беглом взгляде на представленную таблицу можно оценить вероятные проблемы, которые возникнут при несоблюдении правил. Самое безобидное, что может случиться, — это получение параметров из стека не в том порядке, который требуется (при условии, что размеры операндов совпадают). Пример такого несовпадения рассмотрен ниже.

Гораздо хуже будут обстоять дела, если нарушены соглашения очистки стека. Допустим, вызывающий код предполагает размещение операндов в стеке, вызов функции и последующую очистку стека (варианты 1 и 2), а вызываемая функция **Func** также предполагает очистку стека перед передачей управления (варианты 3 и 4). В результате будет нарушен доступ.

Замечание.

В случае запуска debug-версии в среде VC++ подобные ошибки легко отлавливаются специальными проверочными модулями наподобие **chkesp.c**. При обнаружении ошибки на экране появится окно с предупреждением о ней. При этом даже будет выдано указание на возможную ошибку — несоответствие **calling conventions**.

Если вы используете среду, в которой эти правила всегда одни и те же, — все замечательно. Но если среды подготовки DLL и EXE различаются, EXE будет действовать по одним правилам при очистке стека, а DLL — по другим. В результате произойдет нарушение общего доступа к памяти,

а также немедленное аварийное завершение приложения.

Рассмотрим небольшой пример.

Клиент, написанный на VC++:

```
void main()
{
   ...
   // определяем при помощи typedef новый
   // тип - указатель на вызываемую функцию.
   // Очень важно знать типы и количество
   // аргументов, а также тип возвращаемого
   // результата
   // !!! КЛЮЧЕВОЙ МОМЕНТ - __STDCALL !!!
   typedef int (__stdcall *PGetSum)(const int,
   const int);
   // пытаемся получить адрес функции getSum
   PGetSum pGetSum = (PGetSum)GetProcAddress
   (hModule, "getSum");
   // проверяем успешность получения адреса
   _ASSERT(pGetSum != NULL);

   // используем функцию так, словно мы сами
   // ее написали
   const int res = pGetSum(10, 20);
   ...
}
```

При этом DLL, написанная на Delphi 6, предполагает экспорт следующей функции:

```
////////////////////////////////////////
// !!! КЛЮЧЕВОЙ МОМЕНТ - PASCAL!!!
function getSum(const n1: integer; const n2:
integer): integer; pascal;
var
   res: integer;
begin
   res := n1 + n2;

   // сохраняем результат в экспортируемой
   // переменной
   g_N := res;

   // вернуть результат
   Result := res;
end;
```

В данном случае при вызове функция **getSum** получит аргументы в обратном порядке. Так как их размеры совпадают, ничего страшного не произойдет (если учесть, что операция сложения обладает свойством коммутативности). Однако будьте внимательны: подобные ошибки не всегда легко отлаживаются, а совершаются часто.

Проблема 3. Несоответствие моделей управления памятью

Приведем классический пример кода.

Пусть DLL содержит две функции для инициализации блока динамически выделяемой памяти и передачи ее приложению:

```
// malloc/free
void* GetBuffer1(const int size)
{
   return malloc(size);
}

// new/delete
void* GetBuffer2(const int size)
{
   return new char[size];
}
```

Тогда приложение VC++ может использовать эту память примерно следующим образом:

```
void main()
{
   const int BUF_SIZE1 = 5;
   const int BUF_SIZE2 = 8;

   // инициализация памяти при помощи функций
   // из DLL
   char* pBuf1 = (char*)GetBuffer1(BUF_SIZE1);
   char* pBuf2 = (char*)GetBuffer2(BUF_SIZE2);

   // работа с памятью_1
   ...
   // работа с памятью_2
   ...

   // очистка памяти при помощи функций C/C++
   free(pBuf1);
   delete[] pBuf2;
}
```

А приложение Delphi сделает это так:

```
const
   BUF_SIZE1 = 5;
   BUF_SIZE2 = 8;

type
   PChar = ^char;

var
   p1, p2: PChar;
   i: integer;
   ch: char;

begin
   // получить память для работы при помощи
   // функций из DLL
   p1 := GetBuffer1(BUF_SIZE1);
   p2 := GetBuffer2(BUF_SIZE2);

   // работа с буфером_1
   ...
   // работа с буфером_2
   ...

   // освобождаем память при помощи функций DELPHI
   FreeMem(p1);
   FreeMem(p2);
end.
```

Будет ли блок работать? Неизвестно. В данном случае поддержка выделения динамической памяти обеспечивается библиотекой времени исполнения C/C++ (**runtime library**), о которой приложение Delphi не знает. При этом только RTL знает, какой именно адрес она передала для работы. Она также хранит информацию о том, как именно затем этот блок безопасно удалить.

Таким образом, можно предположить, что приведенный фрагмент кода будет работать на VC++ (так как и DLL написана на этом языке, а значит, использует одну и ту же версию RTL), а для Delphi это работать не будет.

Ну, что ж, если так, приведем результаты испытаний:

- в случае исполнения в среде VC++ (принимались настройки проекта по умолчанию): фрагмент кода выдает debug-ошибку о несоответствии хипов;
- в случае исполнения в среде Delphi: среда выдает **runtime error**.

При этом оба приложения выдавали ошибку на первом из двух вызовов освобождения памяти.

Что касается Delphi — еще раз повторимся: Delphi, скорее всего, ничего не знает о том, что такое RTL C/C++, и тем более при освобождении памяти посредством **FreeMem** предполагает, что она была выделена по правилам RTL Delphi. Но это достаточно просто было предугадать.

Почему же в таком случае не работает первый вариант? Дело в том, что функция **GetBufferX** при настройках по умолчанию использует статическую линковку с RTL C/C++ (отсюда и появившиеся примерно 80 «лишних» килобайт кода в случае подготовки кода в release-версии). И DLL при настройках по умолчанию использует также статическую линковку с RTL. Таким образом, они работают с двумя абсолютно независимыми версиями данных библиотек. А те, в свою очередь, выделяют разные области динамической памяти для работы в приложении и в DLL (так называемые локальные кучи).

При этом функция **GetBufferX**, работая в контексте DLL, выделяет область памяти в куче под № 1. А приложение, работая в своем контексте, пытается освободить этот блок в куче под № 2. Отсюда и неочевидные проблемы.

Замечание.

DLL, будучи спроецированной на адресное пространство вызывающего процесса (в данном случае — приложения), практически полностью теряет свою индивидуальность, а следовательно, работает всегда от имени процесса. Употребленное выше выражение «работает в контексте» относится исключительно к тому, с какой библиотекой RTL происходит работа в конкретный момент времени.

Как же заставить этот код работать?

Если мы хотим обеспечить подобную совместимость только в версии VC++ (и DLL, и приложение пишутся в этой среде), то сделать это все-таки можно.

При компиляции RTL с ключом «**Debug Single Threaded**», «**Debug Multithreaded**» приложение и DLL, написанные на VC++, отказывались работать в отладочной версии.

Забегая немного вперед, отметим, что единственно возможным вариантом оставалась компиляция с параметром «**[Debug] MultithreadedDLL**» (**Project Options->C/C++-> CodeGeneration->Use run-time library**). Причем этот ключ должен быть использован при компиляции как EXE, так и DLL. В этом случае и приложение, и DLL пользовались одним и тем же экземпляром библиотеки времени исполнения, а следовательно, обращались к одним и тем же функциям занятия и освобождения памяти. Это, в свою очередь, приводило к тому, что память извлекалась из одной кучи (**heap**).

Однако существует более универсальный и удобный способ, который заставит нашу DLL одинаково хорошо работать в любой среде программирования. Достаточно добиться того, чтобы все действия с манипулированием (регламентированным управлением — инициализацией и освобождением) памятью происходили в одном месте — либо в DLL, либо в EXE: если у нас есть функция **GetBufferX**, то должен быть и **FreeBufferX**, который всегда точно знает, каким именно образом надо освобождать память.

Добавим в нашу DLL функции освобождения памяти:

```
...
// функция освобождения памяти, выделенной при
// помощи GetBuffer1
void FreeBuffer1(void* buf)
{
   free(buf);
}

// функция освобождения памяти, выделенной при
// помощи GetBuffer2
void FreeBuffer2(void* buf)
{
   delete[] (char*)buf;
}
```

Приложения всегда смогут использовать такую DLL, несмотря на то, какая версия RTL используется у конечного пользователя.

В случае VC++:

```
void main()
{
   ...
   // инициализация памяти
   char* pBuf1 = (char*)GetBuffer1(BUF_SIZE1);
   char* pBuf2 = (char*)GetBuffer2(BUF_SIZE2);

   // работа с памятью_1
   ...
   // работа с памятью_2
   ...

   FreeBuffer1(pBuf1);
   FreeBuffer2(pBuf2);
}
```

В случае Delphi:

```
...
begin
   // получить память для работы
   p1 := GetBuffer1(BUF_SIZE1);
   p2 := GetBuffer2(BUF_SIZE2);

   // работа с буфером_1
   ...
   // работа с буфером_2
   ...

   // освобождаем память
   FreeBuffer1(p1);
   FreeBuffer2(p2);

end.
```

Это надолго избавит вас и пользователей написанной вами DLL от подобных проблем.

Проблема 4. Несоответствие типов в средах разработки

Старайтесь применять экспорт функций исключительно с простыми типами аргументов. Как было сказано ранее, каждый компилятор, разбивая класс на удобные и понятные для него компоненты и собирая их обратно, использует свои правила, которые, разумеется, не обязаны совпадать. Кроме того, не всегда возможно обеспечить взаимную эквивалентность различных сложных типов, как в случае простых типов (например, C++ **int** всегда понятен **integer** из **Object Pascal**).

Средства языков **Object Pascal** и C++ во многом обеспечивают такую эквивалентность в той части, которая касается общеупотребительных типов.

С классами все обстоит гораздо сложнее. Такая эксплуатация может быть не всегда корректной даже при условии использования одной и той же среды, как при реализации DLL.

Следует соблюдать осторожность при экспорте классов. На наш взгляд, для системных и общеупотребительных DLL такое вообще недопустимо. Ведь класса той же STL (**standard template library**) может вообще не оказаться у клиентской стороны. Что делать для задач, которые все же требуют использования STL? Мы предпочитаем не экспортировать этот код в DLL, а подключать явно (в статической линковке). Либо стоит идти по сложному пути, связанному с различного рода приемами и ухищрениями, которые, правда, обходятся дополнительными затратами по времени из-за промежуточных перекачиваний массивом STL в T* и обратно!

Техника предполагается следующая. Допустим, есть некоторая общеупотребительная во всех проектах функция, которая должна возвращать массив переменной длины **std::vector <int>**.

Как ее лучше всего экспортировать? Следует:

- переписать интерфейс функции с использованием int*. Если полная адаптация невозможна, то перед возвратом значений создавать буфер, в который переписывать int-значения массива;
- расширить интерфейс функции. Ввести вспомогательные функции, которые возвращают элемент по его индексу (или подмассив элементов). Правда, и в этом случае возникнут дополнительные затраты, связанные с вызовом данной функции.

Стремление обеспечить как можно большее сокрытие деталей реализации еще не раз поможет вам. Клиенту некогда разбираться, каким образом реализован тот или иной механизм, да ему и незачем это делать. Излишние знания в этой области только добавят массу лишних проблем. Ведь DLL — очень мощное средство повторного использования кода. Использовать его надо с умом и осторожностью!

Замечание.

Убедиться в том, что подобное возможно, легко при использовании системных DLL **Windows API**, которые написаны на одном языке программирования и в одной среде разработки (и вряд ли это был **C++ Builder**), а применяются многими.

Полученные результаты представим в виде табл. 2. При этом во всех реализациях предполагался экспорт функции и переменной из DLL. Функция экспортировалась с соглашением вызова **stdcall**. Приложение пыталось импортировать их из DLL. При импорте функции предполагалось использование **calling conventions**, установленного как **stdcall**.

Таким образом, можно сделать вывод, что заявленные возможности четко поддерживаются указанными компиляторами. При этом среда VC++ поддерживает полнофункциональную работу (в том числе с возможностью отложенной загрузки/выгрузки).

Delphi накладывает ряд ограничений, в связи с чем работа с DLL отложенной загрузки невозможна, а при работе с переменными возможен только экспорт. Необ-

Таблица 2

Варианты реализации и использования DLL- и EXE-файлов

Вариант реализации и использования	Результаты
1. DLL: VC++ Приложение: Delphi	
1.1. Неявная загрузка	Импортирование функции обходится без проблем. Импортировать переменную не удалось. Связано это с тем, что Delphi не поддерживает импорт глобальных переменных из DLL.
1.2. Явная загрузка	Импорт функции обходится без проблем. Импорт переменной также обходится без проблем.
1.3. Отложенная загрузка	Не поддерживается Delphi 6
2. DLL: Delphi Приложение: VC++	
2.1. Неявная загрузка	Импорт функции обходится без проблем*. Импорт переменной обходится без проблем*.
2.2. Явная загрузка	Импорт функции обходится без проблем. Импорт переменной также обходится без проблем.
2.3. Отложенная загрузка	Импорт функции обходится без проблем. При этом замечательным образом работает как отложенная загрузка, так и отложенная выгрузка. Импорт переменной в случае использования отложенной загрузки невозможен!

* Выражение «без проблем» означает наличие минимальных препятствий, преодоление которых не составит никакого труда.

ходимо обеспечить функциональность библиотеки таким образом, чтобы избежать прямого обращения с такой переменной, — например, завести функции установки и получения значения (функции аксцессоры). При этом область возможного использования DLL, построенной подобным образом, будет значительно шире. Повысится также безопасность использования кода.

Итак, первое, что бросается в глаза: эти проблемы возникли при работе только с неявной загрузкой. Как можно получить lib-файл для DLL, написанной на Delphi? Ведь по умолчанию компилятор создает только файл с расширением *.dll. Оказывается, это не так сложно, если знать правильные подходы. Воспользуемся для этого рядом утилит, которые входят в состав **Visual Studio**.

Первая из них — **lib.exe** — позволяет на основе DEF-файла и построенной DLL получить lib-файл. Это именно то, что нам необходимо.

Вторая — **impdef.exe** (доступная счастливым обладателям **C++ Builder**) — позволяет на основе DLL-файла сгенерировать DEF-файл.

Замечание.

В качестве примера приведем возможный вариант полученного DEF-файла, сгенерированного **impdef.exe**:

```
LIBRARY   XDLL.DLL

EXPORTS
   _g_N          = g_N          ; g_N
   _getSum       = getSum       ; getSum
```

Эта утилита проста в эксплуатации. Информацию о ключах можно получить из справочной службы **C++ Builder**.

Итак, для нашей DLL мы создали следующий DEF-файл.

Файл **XDll.def**:

```
EXPORTS
      getSum
      g_N
```

Далее запускаем утилиту **lib.exe** с параметрами **/DEF:XDll.def /MACHINE:x86**.

В результате получается lib-файл, который, как и прежде, можно проанализировать при помощи **dumpbin.exe**:

```
Dump of file XDll.lib

File  Type: LIBRARY

      Exports

           Ordinal     name

                       _g_N
                       _getSum

   Summary

         BD .debug$S
         14 .idata$2
         14 .idata$3
          4 .idata$4
          4 .idata$5
          A .idata$6
```

Как видим, не все так замечательно, как можно было бы ожидать. От декорирования избавиться так и не удалось. Но это очень сильно связано с тем, что DLL была сгенерирована **Borland Delphi**. Впрочем, эти проблемы решаются.

Напомним файл клиентского приложения, использующего такую DLL:

```
#include <iostream>

#pragma comment(lib, "XDll.lib")

__declspec(dllimport) int __stdcall getSum
(const int, const int);
__declspec(dllimport) int  g_N;

void main()
{
    const int res = getSum(10, 20);

    // используем функцию так, словно мы сами ее
    // написали
    std::cout<<"getSum(10, 20):"<<res<<std::endl;

    // получим накопленный результат через
    // экспортируемую переменную
    std::cout<<"g_N:"<<g_N<<std::endl;
}
```

При попытке компиляции данного приложения вы получите ошибку о невозможности разрешения имен. Внимательно изучаем это сообщение:

```
Linking...
main.obj : error LNK2001: unresolved external
symbol
    "__declspec(dllimport) int g_N" (__imp_?g_
    N@@3HA)
main.obj : error LNK2001: unresolved external
symbol
    "__declspec(dllimport) int __stdcall getSum
    (int,int)" (__imp_?getSum@@YGHHH@Z)
Debug/XDllClient6.exe : fatal error LNK1120: 2
unresolved externals
Error executing link.exe.

XDllClient6.exe - 3 error(s), 0 warning(s)
```

Как видим, линкер при заданных соглашениях вызова предполагает наличие в библиотеке импорта несколько иных названий функций, нежели те, которые указали в DEF-файле мы (как уже неоднократно было сказано, связано это с декорированием имен).

Подправляем файл **XDll.def**:

```
EXPORTS
      ?getSum@@YGHHH@Z
      ?g_N@@3HA
```

Заметьте: мы убрали префикс вида «__imp». Вероятно, данный префикс применяется для указания некоторой служебной информации, использующей VC++.

Замечание.

Это тесно связано с форматом PE-файла и так называемыми «шлюзами». На самом деле при генерации вызова импортируемой функции компилятор генерирует код:

```
call      SomeFunc
SomeFunc:
jmp       ds:_imp_SomeFunc
```

при этом **ds:_imp_SomeFunc** находится в секции импорта:

```
.idata    _imp_SomeFunc: DWORD
```

Этот-то DWORD загрузчик ОС и заполняет конкретным значением вычисленного (Base+RVA) адреса импортированной функции при явной загрузке DLL.

Повторяем операцию. Приложение вылетает с ошибкой «Не могу найти точку входа для функции …». Немного поразмышляем. Информация, выданная **dumpbin.exe** для поля «**ordinal**», содержала пустые поля. Возможно, дело в этом? Тогда будет предполагаться экспорт функции по ее порядковому номеру. Если компоновщик не может найти имя, наверное, он сможет осуществить подобную связь по порядковому номеру?

Замечание.

Подобные игры с порядковыми номерами необходимы в случае использования Visual C++ версии 6.0. Для седьмой версии написанное ниже не является столь актуальным, потому что неявное связывание замечательно проходит и при отсутствии порядковых номеров в lib-файле.

Попробуем подправить DEF-файл:

```
EXPORTS
    ?getSum@@YGHHH@Z @2
    ?g_N@@3HA @1
```

На этот раз DLL, написанная на Pascal, используется в VC++ в случае неявной загрузки (как было сказано, и без лишних проблем). Даже появляющийся на экране результат не кажется уже таким надоевшим:

```
getSum(10, 20): 30
g_N: 30
```

При использовании варианта с «extern "C"» DEF-файл необходимо представить несколько в ином виде, так как в этом случае предполагается несколько иная стратегия декорирования:

```
EXPORTS
    getSum@8 @2
    g_N @1
```

Использование в программе:

```
extern "C" __declspec(dllimport) int __stdcall
getSum(const int, const int);
extern "C" __declspec(dllimport) int g_N;
```

Замечание (*только для Visual C++ 6.0*).

Как оказалось, ключевое значение в обоих примерах DEF-файла играет правильное использование поля «**ordinal**». Если что-нибудь перепутаете (или просто их не укажете), клиентское приложение откажется работать: при запуске получите сообщение об ошибке «Не могу найти точку входа для функции...». Кстати, это касается всех случаев работы в VC++ с неявной загрузкой (независимо от производителя DLL). В качестве эксперимента мы попробовали провести такие же тесты с DLL, скомпилированной средой MsDev: скомпилировали DLL, а затем при помощи **lib.exe** получили файл импорта (не указав значения порядковых номеров). Результат аналогичный. Получается, что при неявной загрузке по умолчанию связывание происходит по информации именно из этого поля. А вот связывание по имени осуществить не удается. Это каким-то мистическим образом связано с появлением знаков подчеркивания в генерируемом **lib.exe** файле.

Использование DLL, созданных в различных средах программирования

Как-то на форуме **Progz Ru** появился вопрос: можно ли (и если да, то как?) применять DLL в отличной среде программирования, нежели та, которая использовалась при ее создании?

Общая идея того, как это можно реализовать, будет подробно изложена в данном подразделе.

Итак, давайте создадим DLL в среде **Delphi**.

Вот исходный код проекта DLL:

```
library TestLib;

uses
   SysUtils,
   Classes;

function MyFunc (FirstParam:  integer;
                 SecondParam:  PChar;
                 ThirdParam:   PChar;
                 FourthParam:  boolean
                 ): PChar; stdcall; export;

   const CRLF = #13#10;
   var s1 : string;

begin
   if FourthParam then s1 := 'true' else s1 :=
   'false';
   // или IntToStr(integer(FourthParam))

   Result := PChar
   ('Parameters:'+CRLF+
     '   First:      '+IntToStr(FirstParam)+CRLF+
     '   Second:     '+SecondParam+CRLF+
     '   Third:      '+ThirdParam+CRLF+
     '   Fourth:     '+s1);
end;

exports
  MyFunc;

begin
end.
```

Как видите, данная DLL экспортирует функцию **MyFunc**, принимающую параметры различного типа.

Замечание.

Обратите внимание на использование ключевого слова **stdcall** — это позволит без особых проблем вызывать функции этой DLL из программ, написанных на других языках и, возможно, использующих по умолчанию отличные соглашения вызова. В дополнение к сказанному можно привести ссылку из справочной службы системы **Delphi**: «Если вы хотите, чтобы функции вашей библиотеки были доступны в программах, написанных на других языках, лучший способ — использовать stdcall-соглашение вызова в объявлении экспортируемой функции. Другие языки могут не поддерживать соглашение вызова Pascal, которое среда Delphi использует по умолчанию».

Теперь попробуем загрузить созданную DLL и вызвать функцию **MyFunc** в среде **Visual Studio**, используя средства явной загрузки.

Создадим простое консольное приложение, текст которого представлен ниже.

```
// TestDelphiDLL.cpp : Defines the entry point
// for the console application.
//

#include "stdafx.h"
#include <stdio.h>
#include <conio.h>

int main()
{
   HMODULE hdll = ::LoadLibrary("TestLib");
   if(hdll)
   {
      typedef char* (__stdcall *MyFuncType)(int,
      char*,char*,BOOL);

      MyFuncType f = (MyFuncType)::GetProcAddress
      (hdll, "MyFunc");
      if(!f)
         printf("Да гранаты у него не той системы!\
         n");
      else
      {
         char* s = f( 25, "First string", "Second
         string", TRUE );
         printf( "%s\n", s );
      }
      ::FreeLibrary(hdll);
   }
   else printf("DLL not found... Does it exist?\
   n");

   return 0;
}
```

Вывод программы:

```
Parameters:
First: 25
Second: First string
Third: Second string
Fourth: true
```

Для неявной загрузки DLL потребуются дополнительные действия. Дело в том, что форматы lib-файлов, используемых для статической загрузки компаниями Microsoft и Borland, различаются. Поэтому прежде всего необходимо получить правильный lib-файл для среды **Visual C++**.

Используем утилиту **impdef.exe**, получим из DLL список экспортируемых параметров:

```
LIBRARY   TESTLIB.DLL
EXPORTS
   MyFunc       @1    ; MyFunc
```

Изменяем экспортированное имя так, чтобы оно было понятно компилятору VC++:

```
LIBRARY   TESTLIB.DLL
EXPORTS
   MyFunc@16    @1    ; MyFunc
```

Запускаем утилиту **lib.exe**, чтобы получить соответствующий lib-файл.

DLL готова к использованию методом неявной загрузки.

Переписываем код клиента:

```
// TestDelphiDLL.cpp : Defines the entry point
// for the console application.
//
#include "stdafx.h"
#include <stdio.h>
#include <conio.h>
extern "C" _declspec(dllimport) char* __stdcall
MyFunc(int,char*,char*,BOOL);
int main()
{
   char* s = MyFunc( 25, "First string", "Second
   string", TRUE );
   printf( "%s\n", s );
   printf("Press any key...");
   _getch();
   return 0;
}
```

Остается собрать и запустить проект.

Сборка и распространение дистрибутива

При распространении дистрибутива вашей программы (под программой в данном случае понимается любой исполняемый PE-файл — как *.EXE, так и *.DLL) — в том случае, если вашей программой будет пользоваться кто-либо еще, вы должны включить в комплект поставки все необходимые файлы. Имена явно загружаемых вашей программой библиотек вы знаете, в случае неявного подключения библиотек (**implicit linking**) можете легко пропустить один из модулей.

Для отслеживания взаимозависимостей между модулями используйте, например, утилиту **Dependency Walker** (**depends.exe**) из **Platform SDK** или утилиты аналогичного назначения других фирм. Помните: если вы не включите в комплект поставки все необходимые модули, ваша программа может не запуститься на компьютере пользователя или же работать неправильно. Тестируйте ваш комплект поставки на разных машинах.

Наконец, не забывайте о возможных различиях версий поставляемых вами библиотек и уже установленных в системе пользователя. Все пользователи очень болезненно реагируют, если после инсталляции вашей программы уже установленные на компьютере пользователя программы начинают работать неправильно, наблюдаются ошибки и сбои. Соблюдайте следующее правило: все необходимые для работы вашей программы библиотеки необходимо копировать в тот каталог, в который устанавливается ваша программа, ничего не следует трогать в системном или общем каталоге **Windows**.

Экспорт и импорт функций и переменных из EXE-файлов

Немногие программисты знают о том, что функции и переменные можно экспортировать не только из DLL-, но и из EXE-файла. В самом деле, ведь формат DLL- и EXE-файлов одинаков — это обычный **Portable Executable** (PE) файл, и различаются они только одним-единственным битом в поле **Characteristics** заголовка. А раз так, то ничто не мешает нам экспортировать функции и переменные из EXE-файла точно так же, как и из DLL: компоновщик создаст в EXE-файле секцию экспорта и запишет в нее экспортируемые имена. Более того, точно так же, как и для DLL, компоновщик создаст lib-файл, который можно использовать для неявного связывания с другими модулями.

Импорт функций и переменных из EXE-файла не отличается от импорта их из DLL — достаточно подключить к проекту соответствующий lib-файл для неявного связывания.

Поясним сказанное примером. Для простоты создадим консольное приложение со следующим кодом:

```
// TestExport.cpp : Defines the entry point for
// the console application.

#include <stdlib.h>
#include <stdio.h>
#include <windows.h>

extern "C" _declspec(dllexport) int TestFunc()
{
   printf("Hello from TestFunc!\n");
   return 0;
}

extern "C" _declspec(dllexport) DWORD MyVar = 0;

int main()
{
   return 0;
}
```

Наш исполняемый файл (*.EXE) будет экспортировать одну функцию — **TestFunc** и одну переменную — **MyVar** с начальным значением 0. Откомпилируем и соберем этот проект, в результате кроме исполняемого файла получим еще библиотечный файл — **TestExport.lib**.

Теперь создадим проект тестовой DLL (и, разумеется, не забудем подключить к проекту файл **TestExport.lib**):

```
// TestExportDLL.cpp : Defines the entry point
// for the DLL application.

#include <windows.h>

BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD ul_reason_for_call,
                       LPVOID lpReserved
                                          )
{
   return TRUE;
}

extern "C" __declspec(dllexport) DWORD Func1
(DWORD i);
extern "C" __declspec(dllimport) void TestFunc();
extern "C" __declspec(dllimport) DWORD MyVar;

DWORD Func1(DWORD i)
{
   TestFunc();
   MyVar = i+12;
   return (i+5)*10;
}
```

Откомпилируем и соберем тестовую библиотеку. Кстати, наша DLL экспортирует функцию **Func1**, которая и использует импортированные из EXE-файла функцию **TestFunc** и переменную **MyVar**.

Теперь можно дописать в приложении недостающий код и снова собрать его (разумеется, не забудем подключить к проекту файл **TestExportDLL.lib**):

```
extern "C" __declspec(dllimport) DWORD Func1
(DWORD);

int main()
{
   Func1(8);
   printf("After Func1 call MyVar == %d\n",MyVar);
   return 0;
}
```

Запустив тестовое приложение, получим следующий консольный вывод:

```
Hello from TestFunc!
After Func1 call MyVar == 20
```

Однако на практике все не так просто, как в приведенном примере. На пути экспорта функций и переменных из EXE-файла (и импорта их куда-либо) программиста подстерегают несколько подводных камней, о которых расскажем ниже.

Первый подводный камень.

При загрузке любого модуля (EXE или DLL) системный загрузчик передает управление вовсе не функции **main**, **WinMain** или **DllMain** либо на любую другую первую выполняемую функцию приложения (так называемую точку входа), как думают многие программисты, а на специальную, скрытую от программиста точку входа — например, в системе **Microsoft Visual C++** она называется **_WinMainCRTStartup** или **_DllMainCRTStartup**. Задача этой скрытой функции — инициализация библиотеки времени выполнения (CRT) модуля и выполнение некоторых других подготовительных действий (например, вызов конструкторов глобальных объектов). Только после этого она передает управление фунции **main**, **WinMain**, **DllMain** или другой подобной.

Для «обычных» DLL этот процесс выполняется автоматически загрузчиком операционной системы независимо от того, загружается DLL явно (вызовом функции **LoadLibrary**) либо неявно. Если же попробуете загрузить EXE-модуль, экспортирующий что-либо, — с использованием явной или неявной загрузки функцией **LoadLibrary**, — то, разумеется, ни инициализация библиотеки времени исполнения, ни инициализация (конструирование) глобальных объектов выполнены не будут. В результате многие критические для выполнения глобальные структуры данных останутся неинициализированными, в случайном, непредсказуемом состоянии. Последствия этого для программы нетрудно предугадать.

Второй подводный камень.

Это так называемые «таблицы перемещаемых элементов» (relocations). Как правило, любые более-менее серьезные приложение или библиотека содержат многократно вложенные вызовы функций, обращения к переменным в памяти и т. п. В машинном коде, разумеется, все эти вызовы функций, обращения к памяти и т. п. транслируются в относительные или абсолютные адреса, порождая код вида:

```
call 10002B48h
mov eax, [10CF5704h]
```

И так далее. Как правило, для DLL-библиотек эти адреса рассчитываются линкером в предположении, что адрес загрузки DLL совпадает с базовым (а он, как вы помните, для большинства DLL по умолчанию равен 0x10000000).

Конечно, разработчики компляторов понимают, что реально DLL-библиотека может быть загружена по любому другому адресу, отличному от базового. В этом случае конечно же все столь старательно рассчитанные линкером адреса «съедут», и библиотека окажется неработоспособной. Именно поэтому в DLL имеется специальная таблица перемещаемых элементов — в ней перечислены все адреса, подлежащие корректировке. При загрузке системный загрузчик «обходит» все элементы, перечисленные в таблице перемещений, и корректирует соответствующие адреса так, что DLL-библиотека работает корректно.

В EXE-модуле эта таблица перемещаемых элементов отсутствует, а все адреса рассчитываются линкером из предположения, что EXE-модуль всегда загружается по одному и тому же фиксированному базовому адресу (а для Win-приложений он обычно равен **0x00400000**). Что же произойдет, если попытаться загрузить EXE-модуль, как DLL, функцией **LoadLibrary**? Поскольку таблицы перемещений нет, никакие адреса

скорректированы не будут, и после загрузки по какому-то известному одному только загрузчику ОС произвольному адресу (а адрес 0x00400000 уже занят, по нему загружено «первичное» EXE-приложение — то самое, которое и пытается загрузить наш EXE-модуль как DLL) все адреса вызовов и обращений к памяти будут показывать «пальцем в небо».

Получение доступа к неэкспортируемым функциям и переменным DLL

Иногда, хотя и очень редко, программисту все же требуется получить доступ к локальным (неэкспортируемым) функциям и переменным чужой DLL.

Конечно, если исходные тексты DLL доступны, достаточно объявить соответствующие функции экспортируемыми и перекомпилировать модуль. Но что делать, если исходные тексты недоступны (например, разработчик этой DLL уволился)? Конечно, решение есть. Для этого, правда, потребуется дизассемблировать DLL и вычислить смещения точек входа искомых функций и переменных в файле.

Замечание.

Хотя мы используем дизассемблирование кода только в исследовательских целях, однако Закон РФ от 23 сентября 1992 года № 3523-1 «О правовой охране программ для ЭВМ и баз данных» в статье 15 содержит следующее прямое разрешение на выполнение дизассемблирования:

«Лицо, правомерно владеющее экземпляром программы для ЭВМ, вправе без согласия правообладателя и без выплаты дополнительного вознаграждения декомпилировать или поручать декомпилирование программы для ЭВМ с тем, чтобы изучать кодирование и структуру этой программы при следующих условиях:

- информация, необходимая для взаимодействия независимо разработанной данным лицом программы для ЭВМ с другими программами, недоступна из других источников;
- информация, полученная в результате этого декомпилирования, может использоваться лишь для организации взаимодействия, независимо от разработанной данным лицом программы для ЭВМ с другими программами, а не для составления новой программы для ЭВМ, по своему виду существенно схожей с декомпилируемой программой для ЭВМ или для осуществления любого другого действия, нарушающего авторское право;
- декомпилирование осуществляется в отношении только тех частей программы для ЭВМ, которые необходимы для организации такого взаимодействия».

Данный закон прекратил свое действие с 1 января 2008 года со вступлением в действие четвертой главы Гражданского кодекса РФ. Однако указанные положения теперь содержатся в статье 1280 ГК РФ.

Вспомним, что PE-файл представляет собой образ памяти, линейно отображаемый на адресное пространство загружающего процесса. Возвращаемое функцией **LoadLibrary** значение **HINSTANCE** есть не что иное, как базовый адрес, по которому PE-файл отображается на адресное пространство процесса. Теперь, зная смещение точки входа нужной функции в PE-файле (**RVA — relative virtual address**), легко рассчитать адрес точки входа в адресном пространстве процесса. Последующее использование рассчитанного таким образом адреса не отличается от использования адреса, возвращенного функцией **GetProcAddress**. (Собственно, функция **GetProcAddress** делает все ту же высокоинтеллектуальную работу, но скрытно от программиста. Смещение точки входа извлекается не из дизассемблированного кода DLL, а из таблицы экспорта.)

Поясним сказанное на примере. Пусть, например, дизассемблер (в нашем случае — IDA) показал, что адрес нужной функции — 0x10001BB4.

```
.text:10001BB4  ; ------------------- S U B R O U T I N E -------------------------
.text:10001BB4
.text:10001BB4  ; Attributes: bp-based frame
.text:10001BB4
.text:10001BB4  sub_10001BB4      proc near     ; CODE XREF: sub_10001F8C+9
.text:10001BB4
.text:10001BB4  arg_0             = dword ptr   8
.text:10001BB4  lpFileName        = dword ptr   0Ch
.text:10001BB4
.text:10001BB4                    push   ebp
.text:10001BB5                    mov    ebp,   esp
.text:10001BB7                    push   [ebp+lpFileName] ; lpFileName
.text:10001BBA                    push   [ebp+arg_0]      ; int
.text:10001BBD                    call   sub_10001B44
.text:10001BC2                    add    esp,   8
.text:10001BC5                    test   eax,   eax
.text:10001BC7                    jnz    short loc_10001BCD
.text:10001BC9                    xor    eax,   eax
.text:10001BCB                    pop    ebp
.text:10001BCC                    retn
.text:10001BCD ; ---------------------------------------------------------------------
```

Чтобы получить смещение точки входа в файле, из полученного значения вычтем базовый адрес загрузки данной DLL (он, как вы помните, в подавляющем большинстве случаев равен 0x10000000, точно установить его можно с помощью любой подходящей утилиты). Таким образом, получим, что нужное нам смещение равно 0x00001BB4.

Базовый адрес загрузки DLL в адресном пространстве вызывающего процесса можно получить, явно вызвав **LoadLibrary** и приняв возвращенное функцией значение за целое беззнаковое 32-битное число. Это значение может отличаться от базового адреса загрузки (0x10000000), указанного в заголовке DLL, поэтому и необходимо получить его точное значение вызовом **LoadLibrary** (разумеется, после завершения работы с DLL нужно не забыть вызвать **FreeLibrary**). Если искомая DLL уже была спроецирована на адресное пространство процесса, то вызов **LoadLibrary** всего лишь увеличивает значение счетчика использований DLL, не загружая ее вновь, и возвращает адрес, по которому она была загружена (а вызов **FreeLibrary** — уменьшает значение счетчика).

Пусть, например, возвращенное **LoadLibrary** значение равно 0x10050000 — это есть тот реальный адрес, по которому спроецирована наша DLL. Добавив к этому значению смещение 0x00001BB4, получим 0x10051BB4 — это есть адрес точки входа искомой функции в адресном пространстве нашего процесса. Теперь можно вызывать функцию, обращаясь с полученным адресом так же, как если бы он был возвращен функцией **GetProcAddress**. Разумеется, при этом надо не забыть о правильной передаче параметров функции и об очистке стека. В рассматриваемом примере функция вызывается с соглашением вызова **_cdecl**, что хорошо видно при анализе вызывающего кода.

```
.text:10001F8F        push    [ebp+arg_4]
.text:10001F92        push    [ebp+arg_0]
.text:10001F95        call    sub_10001BB4
.text:10001F9A        add     esp, 8
```

Если сможете рассчитать адрес точки входа желаемой функции, то это уже не составит большого затруднения. Кстати, точно так же можно получить доступ и к любой локальной переменной — достаточно только знать ее тип (строго говоря, даже не тип, а размер переменной в байтах).

Экспортирование классов из DLL, созданных различными средами программирования

Есть еще одна проблема при попытке экспорта классов, которая не упоминалась ранее, — использование такого класса в другой среде. Можно сказать, что это практически неосуществимо «честными» способами программирования.

Допустим, мы создали DLL в среде **Visual C++** и экспортируем оттуда нашу функцию **getSum**:

Файл **XDll.h**.

```
//////////////////////////////////////////
#ifdef XDLL_EXPORTS
	#ifdef __cplusplus
		#define XDLL_API extern "C"
		 __declspec(dllexport)
	#else
		#define XDLL_API __declspec
		(dllexport)
	#endif // __cplusplus
#else
	#define XDLL_API __declspec(dllimport)
#endif // XDLL_EXPORTS

//////////////////////////////////////////
XDLL_API int __stdcall getSum(const int n1,
const int n2);
```

Файл **XDll.cpp**.

```
//////////////////////////////////////////
//
XDLL_API int __stdcall getSum(const int n1,
const int n2)
{
	const int n = n1 + n2;
	g_N = n;
	return n;
}
```

Замечание.

Обратите внимание на использование ключевых слов **extern «C»** и **__stdcall**. Именно эти особенности позволят использовать данную библиотеку в другой среде программирования.

Для того чтобы использовать эту функцию в проекте BC++, нужно преобразовать lib-файл, который получен после компиляции в среде VC++. Напомним: все созданные объектные файлы создаются в формате COFF. **C++ Builder**, в свою очередь, предпочитает работать с OMF-форматами. Указанные разногласия можно преодолеть при помощи утилиты **coff2omf**, входящей в состав **Borland C++ Builder**.

Таким образом, подаем на вход этой утилиты lib-файл нашей DLL в формате COFF:

```
...\VC++\XDll\Debug>coff2omf XDll.lib XDll2.lib
COFF to OMF Converter Version 1.0.0.74 Copyright
(c) 1999, 2000 Inprise Corporation
```

На выходе получаем lib-файл, пригодный для среды **C++ Builder**. Подключаем данный файл (**XDll2.lib**) к созданному проекту, в котором необходимо использовать данную DLL: **Project -> Add To Project**.

Вызов функции **getSum** в этом случае по-прежнему тривиален:

```
extern "C" int __stdcall getSum(int, int);

int main()
{
    std::cout < "getSum(10,20): " << getSum(10,20)
     << std::endl;

    return 0;
}
```

Неявная загрузка работает. И это все осуществимо, несмотря на создание DLL в совершенно другой среде. С функциями, использующими соглашение вызова **stdcall**, все действительно так — по такой же схеме созданы стандартные заголовочные файлы.

Рассмотрим для примера файл **WinSock2.h**:

```
#if INCL_WINSOCK_API_PROTOTYPES
WINSOCK_API_LINKAGE
int
WSAAPI
WSAStartup(
    IN WORD wVersionRequested,
    OUT LPWSADATA lpWSAData
    );
#endif /* INCL_WINSOCK_API_PROTOTYPES */
```

А определение символа **WINSOCK_API_LINKAGE** найдем чуть ранее в этом же файле:

```
#ifndef WINSOCK_API_LINKAGE
#ifdef DECLSPEC_IMPORT
#define WINSOCK_API_LINKAGE DECLSPEC_IMPORT
#else
#define WINSOCK_API_LINKAGE
#endif
#endif

#ifdef __cplusplus
extern "C" {
#endif
```

Как видите, техника использования абсолютно аналогичная.

В случае же использования (точнее, попытки использования) классов возникнут практически непреодолимые сложности:

- невозможность использования ключевого слова **extern «C»** для экспорта классов;
- разные соглашения относительно правил использования декорирования имен.

Первая проблема связана с тем, что все имена в этом случае будут задекорированы. Сделано будет это по-разному в случае использования **VC++** и **C++ Builder**. И второе разногласие преодолеть не удастся. Еще раз напомним, что в случае экспорта классов декорирование имен является именно тем способом, который помогает линкеру (в случае неявной загрузки) определить местоположение соответствующих методов и членов-функций (RVA). При различной методике такого декорирования никакими средствами нельзя заставить линкер найти необходимую член-функцию.

К примеру, вот так происходит декорирование методов в случае использования **VC++** (здесь, как и раньше, нам опять поможет утилита **dumpbin** с параметром **/exports**):

```
??0CSummator@@QAE@ABV0@@Z
(public: __thiscall CSummator::CSummator(class
CSummator const &))
??0CSummator@@QAE@H@Z
(public: __thiscall CSummator::CSummator(int))
??1CSummator@@QAE@XZ
(public: __thiscall CSummator::~CSummator(void))
??4CSummator@@QAEAAV0@ABV0@@Z
(public: class CSummator & __thiscall CSummator::
operator=(class CSummator const &))
??_7CSummator@@6B@
(const CSummator::'vftable')
??_FCSummator@@QAEXXZ
(public: void __thiscall CSummator::'default
constructor closure'(void))
?Add@CSummator@@QAEHH@Z
(public: int __thiscall CSummator::Add(int))
?GetBalance@CSummator@@UAEHXZ
(public: virtual int __thiscall CSummator::
GetBalance(void))
?GetDevilSum@CSummator@@SAHXZ
(public: static int __cdecl CSummator::GetDevilSum
(void))
?m_DevilSum@CSummator@@2HA
(public: static int CSummator::m_DevilSum)
```

Аналогичная реализация в **C++ Builder** даст нам следующие результаты. Выполняем команду:

```
impdef XDll.def XDll.dll
```

и смотрим на вывод DEF-файла:

```
@CSummator@$bctr$qxi      ; CSummator::CSummator
                            (const int)
@CSummator@$bdtr$qv       ; CSummator::~CSummator
                            ()
@CSummator@Add$qxi        ; CSummator::Add(const
                            int)
```

```
@CSummator@GetBalance$xqv ; CSummator::GetBalance
                            () const
___CPPdebugHook           ; ___CPPdebugHook
```

Замечание.

В обоих случаях (**VC++** и **C++ Builder**) осуществлялся экспорт следующего класса:

```
//////////////////////////////////////////
// class CSummator
class XDLL_API CSummator
{
public:
   CSummator(const int n = 0);
   ~CSummator();

   int Add(const int n);

   virtual int GetBalance();
   static int GetDevilSum();
   static int m_DevilSum;

private:
   int m_N;
};
```

Количество экспортированных методов определяется логикой реализации компилятора.

Замечание.

Чтобы убедиться в том, что **VC++** не поймет подобных аббревиатур, можно, как и ранее, использовать утилиту **undname** из набора **VC++**. Никакого дедекорирования не произойдет. Например:

```
Microsoft (R) C++ Name Undecorator
Copyright (C) Microsoft Corporation
1981-2001. All rights reserved.

Undecoration of :- "@getSum$qxixi"
is :- "@getSum$qxixi"
```

Существуют два варианта действий:

• использовать явную загрузку — пример такого использования был приведен чуть выше (с объектами, создаваемыми на стеке и в динамической памяти). Подобную технику можно применять без малейших изменений;

• использовать технику виртуальных функций с теми же самыми ограничениями. В этом случае из DLL получаем созданный объект (посредством специально предназначенной для этого функции). Затем используем его виртуальные функции (проблем с декорированием имен в этом случае не будет в связи с одинаковым расположением смещений виртуальных функций в **_vtbl**). После этого уничтожаем объект, вызывая специально созданную для этого функцию в разделе экспорта этой же DLL.

Все это не очень удобно. Поэтому настоятельно советуем избегать использования механизма экспорта классов в DLL. Чтобы окончательно убедить читателя, рассмотрим простой пример, который подтвердит все сказанное выше.

1. Создадим следующий h-файл в среде **Borland C++ Builder**.

```
#ifdef XDLL_EXPORTS
       #ifdef __cplusplus
              #define XDLL_API extern "C"
               __declspec(dllexport)
       #else
              #define XDLL_API __declspec
              (dllexport)
       #endif // __cplusplus
#else
       #ifdef __cplusplus
              #define XDLL_API extern "C"
              __declspec (dllimport)
       #else
              #define XDLL_API __declspec
              (dllimport)
       #endif // __cplusplus
#endif // XDLL_EXPORTS

XDLL_API int __stdcall getSum(const int n1, const
int n2);

// класс с виртуальными функциями благодаря
// использованию #ifdef/#endif/#ifndef-конструкций,
// один и тот же заголовочный файл можно
// использовать в случае как проекта самой DLL,
// так и разработки клиентских приложений
```

```
class CSummator
{
public:
#ifdef XDLL_EXPORTS
	CSummator(const int n = 0);
	~CSummator();
#endif

	// в случае клиентского приложения будем
	// иметь чисто виртуальные функции
	virtual int __stdcall GetBalance() const
	#ifndef XDLL_EXPORTS
		= 0
	#endif
	;

	virtual int __stdcall Add(const int n)
	#ifndef XDLL_EXPORTS
		= 0
	#endif
	;

private:
	int m_N;
};

// специальная функция для конструирования объекта
XDLL_API CSummator* __stdcall InitSummator(const
int n);
// специальная функция для вызова деструктора объекта
XDLL_API void __stdcall ReleaseSummator(CSummator*);
```

2. Соответствующий cpp-файл.

```
// не забываем в случае проекта DLL определить
// соответствующий #define
#define XDLL_EXPORTS
// в этом случае получим определение класса
// с виртуальными (не чисто виртуальными!)
// функциями
#include "XDll6.h"

////////////////////////////////////////////
//
XDLL_API int __stdcall getSum(const int n1, const
int n2)
{
	const int n = n1 + n2;

	return n;
}

////////////////////////////////////////////

CSummator::CSummator(const int n)
{
	m_N = n;
}

CSummator::~CSummator()
{
}

int __stdcall CSummator::GetBalance() const
{
	return m_N;
}

int __stdcall CSummator::Add(const int n)
{
	return m_N += n;
}

// специальная функция конструирования объекта
// для использования механизма виртуальных функций;
// необходимо возвращать указатель на объект
CSummator* __stdcall InitSummator(const int n)
{
	return new CSummator(n);
}

// функция освобождения памяти, выделенной ранее
// функцией инициализации; при этом будет вызван
// соответствующий деструктор удаляемого объекта
void __stdcall ReleaseSummator(CSummator*
pSummator)
{
	delete pSummator;
}
```

Библиотека готова.

3. Теперь нужно получить корректный lib-файл (в формате COFF) для среды **VC++**.

Для начала создадим DEF-файл на основе анализа содержимого DLL:

```
impdef XDll6.def XDll6.dll
```

Затем необходимо подправить полученный DEF-файл. Дело в том, что **impdef** немного «забывает» о том, что было использовано соглашение **stdcall** (в случае работы с библиотекой от **Microsoft impdef** об этом «помнит»).

Исходный DEF-файл, полученный после **impdef**:

```
LIBRARY   XDLL6.DLL

EXPORTS
   InitSummator         @2  ; InitSummator
   ReleaseSummator      @3  ; ReleaseSummator
   ___CPPdebugHook      @4  ; ___CPPdebugHook
   getSum               @1  ; getSum
```

Добавляем к функциям инициализации и уничтожения объекта знак «@», после которого определяем число байтов, необходимых для размещения параметров:

```
LIBRARY   XDLL6.DLL

EXPORTS
   InitSummator@4       @2  ; InitSummator
   ReleaseSummator@4    @3  ; ReleaseSummator
   ___CPPdebugHook      @4  ; ___CPPdebugHook
   getSum@8             @1  ; getSum
```

Теперь используем **lib.exe** для получения корректного lib-файла:

```
lib /def:XDll6.def /machine:x86
```

Содержимое lib-файла, как и раньше, можно просмотреть при помощи утилиты **dumpbin** с ключом **/exports**:

```
Dump of file XDll6.lib

File Type: LIBRARY

     Exports

       Ordinal    name

             2    _InitSummator@4
             3    _ReleaseSummator@4
             4    ____CPPdebugHook
             1    _getSum@8
```

Мы получили именно то, что и должны были получить!

Замечание.

Для просмотра lib-файлов формата OMF можно использовать утилиту **Borland C++ Builder tdump** с ключом **-li**. Например:

```
:\Builder C++\XDll6>tdump -li XDll6.lib
Turbo Dump  Version 5.0.16.12 Copyright (c)
1988, 2000 Inprise Corporation
          Display of File XDLL6.LIB

Impdef: (Name)     XDll6.????=InitSummator
Impdef: (Name)     XDll6.????=ReleaseSummator
Impdef: (Name)     XDll6.????=___CPPdebugHook
Impdef: (Name)     XDll6.????=getSum
```

Более подробную информацию о работе этой утилиты можно получить из справочной службы **C++ Builder**.

4. Вызов класса из **VC++**. Попробуем использовать неявную загрузку и механизм виртуальных функций, чтобы продемонстрировать работу этой техники. Создадим простое консольное приложение.

Файл **main.cpp**:

```
////////////////////////////////////////////
//
#include "XDll6.h"
#include <iostream>

// подключаем lib-файл, который был сгенерирован
// незадолго до этого
// внимание: lib-файл создавался на основе DLL,
// полученной от компилятора компании Borland!
#pragma comment(lib, "XDll6.lib")

int main()
{
   // вызываем экспортированную функцию создания
   // объекта
   CSummator* pSum = InitSummator(10);
   // используем механизм виртуальных функций
   // особое внимание стоит обратить на то, что мы
   // и не думали их экспортировать! Все вызовы
   // виртуальных функций осуществляются посредством
   // смещений в _vtbl
   pSum->Add(20);

   // вызов еще одной виртуальной функции
   std::cout < pSum->GetBalance() <
std::endl;

   // не забываем вызвать функцию уничтожения
   // объекта
   ReleaseSummator(pSum);

   return 0;
}
```

Как видите, все достаточно прозрачно. Вызываем функцию инициализации объекта, чтобы получить указатель на него. Затем используем вызовы виртуальных функций, которые разрешаются посредством специальных смещений в **_vtbl** созданного объекта (а смещения определяются на основе анализа декларации класса). Так как была использована одна и та же декларация, все виртуальные методы будут адресоваться по правильным смещениям. И это совершенно не зависит от того, были эти функции экспортированы ранее или нет.

После этого вызываем специально экспортированную функцию для вызова деструктора созданного объекта и его удаления из памяти.

На вопросе памяти в данном контексте стоит остановиться особо. Можно ли явно записать в коде конструкцию вида

```
delete pSum;
```

Сделать этого нельзя. Разберемся почему.

Прежде всего функция деструктора не объявлена у нас виртуальной. Следовательно, компилятор не знает RVA этого деструктора, чтобы корректно разрешить такой вызов. Если объявить деструктор виртуальным, это все равно не поможет. Потому что, как правило, runtime-библиотеки используют различные механизмы выделения и освобождения памяти.

Даже при использовании одной и той же среды программирования для DLL может применяться другая технология управления памятью, нежели та, которую клиент такой DLL будет применять. Так что даже если и сможем разрешить вызов деструктора, выделенная память, которую получаем из DLL посредством вызова функции **InitSummator**, не будет правильно освобождена. Какие ошибки будут возникать при попытке такого неправильноо освобождения, предсказать невозможно.

Использование STL в DLL

В свое время у авторов данной статьи возникли вопросы относительно возможности использования STL в DLL. Результаты (именно в том виде, в котором они были получены по мере углубления в данную тему) представлены в данном разделе.

Замечание.

Весь приведенный ниже текст относится к реализации STL и CRT **Visual Studio** версии 6.0. Подобных проблем в VC++ 7.0 может не наблюдаться.

При попытке использования STL в DLL возникла неочевидная на первый взгляд проблема.

Построим два проекта: проект DLL, который использует два класса — стандартный класс **vector<>** из библиотеки шаблонов STL и наш собственный класс вектора (**CMyVector**); другой проект представляет собой консольное приложение, которое пользуется услугами данной DLL.

1. Проект DLL. Предполагается, что в DLL определяется класс **CMyVector**, который является подобием реализации класса **vector<>** из STL. Оставлены лишь необходимые методы, которые будут пояснять суть проблемы.

Файл **DLL.h**.

```
#include <vector>
#include <memory>

#ifdef DLL_EXPORTS
#define DLL_API __declspec(dllexport)
#else
#define DLL_API __declspec(dllimport)
#endif

////////////////////////////////////////////
// class MyVector
class DLL_API MyVector
{
public:
   MyVector(const int size);
   ~MyVector();
```

```
    int operator[](const int index); // вернуть
                                     // элемент по индексу
    const int Size(); // вернуть размер
    void Resize(const int new_size);// изменить
                                    // размер
private:
    int* m_Pointer;
    int m_Size;
};

//////////////////////////////////////////
// export 1
DLL_API void func1(MyVector& c1, const int new_
size);
// export 2
DLL_API void func2(std::vector<int>& c2, const
int new_size);
```

Файл **DLL.cpp**.

```
#include "Dll.h"

//////////////////////////////////////////
// MyVector
MyVector::MyVector(const int size)
{
    m_Pointer = new int[size];
    m_Size = size;
}

MyVector::~MyVector()
{
    delete[] m_Pointer;
}

int MyVector::operator[](const int index)
{
    return m_Pointer[index];
}

const int MyVector::Size()
{
    return m_Size;
}

void MyVector::Resize(const int new_size)
{
    delete[] m_Pointer;
    m_Pointer = new int[new_size];
    m_Size = new_size;
}

//////////////////////////////////////////
DLL_API void func1(MyVector& c1, const int new_
size)
{
    c1.Resize(new_size);
}

//////////////////////////////////////////
DLL_API void func2(std::vector<int>& c2, const
int new_size)
{
    c2.resize(new_size);
}
```

2. Проект консольного приложения, которое обращается к экспортированным сервисам DLL. В данном проекте реализована тестовое приложение для работы с Dll.dll.

Файл **main.cpp**.

```
#include "..\Dll\Dll.h"

#include <iostream>

#pragma comment(lib, "..\\Debug\\Dll.lib")

int main(int argc, char* argv[])
{
    // 1
    MyVector v1(100);
    std::cout << "Size of v1-vector:" << v1.Size()
    << std::endl;

    func1(v1,  200);
    std::cout << "Size of v1-vector:" << v1.Size()
    << std::endl;

    v1.Resize(300);
    std::cout << "Size of v1-vector:" << v1.Size()
    << std::endl;

    std::cout <<  std::endl;

    // 2
    std::vector<int> v2(100);
    std::cout << "Size of v2-vector:" << v2.size()
    << std::endl;
```

```
    v2.resize(200);
    std::cout << "Size of v2-vector:" << v2.size()
    << std::endl;

    // !!! будьте предельно внимательны!
    func2(v2, 300);
    std::cout << "Size of v2-vector:" << v2.size()
    << std::endl;

    return 0;
}
```

В первом варианте реализован вызов метода **Resize** класса **CMyVector**. Заметим, что выделение памяти не имеет негативных последствий.

Во втором варианте работаем с **vector** из STL. DLL находится в адресном пространстве процесса. При этом первые два выделения памяти (все происходит от имени процесса) проходят без проблем. Попытка третьего выделения памяти приводит к появлению **assert**.

Предварительно можно сделать следующие выводы.

Во-первых, передача **vector&** в качестве параметра в функцию DLL, изменяющую содержимое этого вектора с реаллокированием памяти, представляется более чем спорной (говоря прямо — принципиально неверной), хотя в стандарте прямого запрета (типа «**ill-formed code**») мы не нашли.

Во-вторых, выполнение **new** и **delete** в различных EXE/DLL-модулях представляется грубой ошибкой. Ведь DLL может использоваться и в совершенно отличной среде. Нетрудно себе представить, к чему может привести такая ситуация:

- память по оператору **new** выделяется в Delphi-приложении, которое использует диспетчер памяти Borland;
- **delete** выполняется в DLL на MSVC, которая полагается на алгоритмы освобождения памяти, принятые в MS, и считает, что память была выделена по MS-алгоритмам.

GPF (**General Protection Fault**) представляется самой малой бедой, которая может приключиться. В худшем случае можно разнести всю кучу (хип) и получить огромное количество практически невоспроизводимых ошибок в программе, причем разных от запуска к запуску.

При первом же прогоне **NuMega BoundsChecker** (**BC**) выдал достаточно подробную информацию о проблеме. Дополнительная информация была получена от отладчика MSVC:

```
HEAP[Console.exe]: Invalid Address specified to
RtlValidateHeap( 012F0000, 01300860 )
```

Не менее интересен тот факт, что в **Release**-сборке все было выполнено без единой помарки. Впрочем, это и ожидалось, поскольку было подозрение, что **RtlValidateHeap** работает только в отладочной версии:

```
Size of v1-vector: 100
Size of v1-vector: 200
Size of v1-vector: 300

Size of v2-vector: 100
Size of v2-vector: 200
Size of v2-vector: 300
```

Пример, конечно, несколько надуманный, но мы стали заниматься этим вопросом исключительно для того, чтобы разобраться с возможными проблемами использования STL в DLL.

Написанный класс (по идее) полностью повторял действия, которые производятся с **vector<>**, — выделение памяти со стороны клиента и последующее выделение со стороны DLL. При этом никаких ошибок не происходит. Разумеется, так просто оставить эту проблему мы не могли.

Следующий факт, который, с нашей точки зрения, заслуживал особого рассмотрения: почему в отладочной версии **assertion failed** (и где именно это **assertion** происхо-

дит), а в релизе эта проверка отсутствует? Ведь на первый взгляд кажется, что собственный класс должен вести себя так же. Однако это не так. Очевидно, сей факт показывает различия в стратегиях выделения и освобождения памяти в отладочной и релизной версиях библиотеки, а также в STL и **C Run-time**. Эти тонкие различия в поведении библиотек необходимо точно знать, чтобы случайно не наступить на грабли в коммерческом проекте.

Какая именно проверка (**assertion**) срабатывает, удалось установить почти мгновенно:

```
/ *
  * If this ASSERT fails, a bad pointer has been
  * passed in. It may be totally bogus, or it may
  * have been allocated from another heap.
  * The pointer MUST come from the 'local' heap.
  */
_ASSERTE(_CrtIsValidHeapPointer(pUserData));
```

Обратим внимание на самое главное: «…*it may have been allocated from another heap*». Вероятно, в этом заключена причина проблем.

После нескольких часов работы с исходниками STL мы пришли к таким выводам (небесспорным):

- несмотря на то что мы всегда используем декларацию вида, например, **vector<int> vec**, на самом деле это декларация вида **vector<int, std::allocator> vec**. Не будем забывать о параметре шаблона по умолчанию, который задает класс распределителя памяти (это верно и для других контейнеров);
- в STL перегружаются **operator new** и **operator delete**, и стандартный аллокатор STL использует перегруженные версии этих операторов, а именно он (**std::allocator**) используется для конструирования объектов STL, изменения размеров и т. п.

Таким образом, использованные в нашем классе **::operator new** и **::operator delete** и использованные вектором **std ::operator new** и **std::operator delete** — это РАЗНЫЕ операторы. Именно этим можно объяснить отсутствие **assertion** в нашем классе и наличие его в стандартном классе вектора.

Мысль относительно использования разных операторов выделения памяти заинтересовала особо. Можно было действовать в двух возможных направлениях:

1) изменить **std::vector::allocator** и оценить реакцию;
2) изменить **allocator** в **MyVector** и оценить реакцию.

После проведенных испытаний на данный момент выяснилось следующее:

1) если объявить вектор с аллокатором по умолчанию, программа сбоит на последнем вызове функции **func2** (как это было продемонстрировано в прошлый раз);
2) если объявить его с каким-либо другим аллокатором, то сбоев не возникает;
3) мы пытались добиться того, чтобы сбой возникал при использовании нашего собственного класса. Достичь этого не удалось, несмотря ни на что: работа точно с таким же аллокатором, как у второго вектора, не приводила к сбоям.

Таким образом, получалось, что разница в стратегиях выделения памяти есть, но эта разница настолько неочевидна, что совмещение стратегий не приводит к сбоям в обоих классах.

Еще раз повторимся: мы изо всех сил пытались сделать так, чтобы начал сбоить наш собственный класс. Например, ввод собственного аллокатора с простым переопределением методов приводит к тому, что и STL-вектор перестает сбоить. Причем интересен факт, что в переопределенных методах просто-напросто вызываются методы базового класса.

И все — никакие дополнительные действия не нужны (см. код **CMyAllocator** ниже).

Файл **Dll.h**.

```
...
//////////////////////////////////////////
// class MyAllocator
class DLL_API MyAllocator: public std::allocator
<int>
{
public:
   pointer allocate(size_type _count, const void*
   Other = NULL);

   void deallocate(pointer _Ptr, size_type _Count);
};
```

Файл **Dll.cpp**.

```
...
//////////////////////////////////////////
// MyAllocator
MyAllocator::pointer   MyAllocator::allocate(size_
type _count, const void* Other)
{
   using namespace std;
   return allocator<int>::allocate(_count, Other);
}

void MyAllocator::deallocate(pointer _Ptr, size_
type _Count)
{
   using namespace std;
   allocator<int>::deallocate(_Ptr, _Count);
}
```

В поисках ответа на вопрос о том, почему наш вектор **MyVector** и **std::vector** ведут себя по-разному при изменении размеров (речь идет пока только о первом варианте теста), мы дизассемблировали и исследовали тестовое приложение и DLL. После этого можно было с уверенностью сказать, что **MyVector** и **std::vector** используют существенно разные механизмы работы с памятью.

1. Первый механизм (в **MyVector**) использует обращение к функциям распределения памяти **C Runtime Library**, отладочная информация для этого механизма в некоторых функциях содержит ссылку на файл **mlock.c**, а этот файл есть не что иное, как механизм многопоточной блокировки Microsoft, имеющий в заголовке «**mlock.c** — **Multi-thread locking routines**». После некоторого исследования этот первый механизм манипуляции памятью можем квалифицировать как системный оператор **new**.

2. Второй механизм работы с памятью используется **std::vector** и никак не пересекается с первым. Во всяком случае, нам не удалось обнаружить совместно вызываемых ими функций и т. п. Оказалось, что этот второй механизм целиком «вшит» в DLL/EXE как статический код. Этот механизм не использует объектов многопоточной синхронизации.

3. Наконец, рапорт BC показывает, что функция API **HeapCreate** вызывается при старте тестового приложения ДВА раза. Наоборот, если исключить из тестового приложения всякое упоминание о STL (что мы и проделали), то **HeapCreate** вызывается ровно (и только) ОДИН раз. Анализ дизассемблированного кода (откуда вызывается **HeapCreate**) показывает, что (при использовании STL) DLL создает собственную кучу, и происходит это в функции **DllMain**.

4. Стек вызовов при обрушении **std::vector** показывает, что приложение сбоит при попытке освобождения «чужого» блока памяти.

Резюме: стартовый код тестовой DLL (выполняемый до входа в функцию **main**) выделяет собственную кучу и создает механизм управления ею, вероятно, для увеличения скорости работы объектов STL [1]. Эта куча STL — потоко-беззащитная. Очевидно, две реализации — **MyVector** и **std::vector** — не только используют разные механизмы выделения памяти, но и выделяют ее из разных куч. Попытка освобождения блока, выделенного в «чужой» куче, заканчивается крахом.

На данный момент ясно следующее:

- при использовании STL в DLL динамическая библиотека создает собственную кучу, которая применяется ею для размещения блоков данных объектов STL;
- с учетом этого при использовании в DLL STL необходимо строго придерживаться правила: создание, разрушение и все действия с объектами STL, которые могут привести к перераспределению памяти (а таких действий довольно много), должны выполняться в одном и том же модуле (EXE/DLL). В противном случае возникнет конфликт разных куч.

Изначально складывалось впечатление, что создание разных куч происходит только в debug-версии. Но рапорт BC показывает, что и в release-версии функция API **HeapCreate** вызывается два раза. А отладочная печать показывает, что и в релизе блоки данных **std::vector** и **MyVector** выделяются из разных куч. Таким образом, стало очевидно, что причина иная: в релизе не выполняется тот самый **assertion**, который срабатывает в отладочной версии, поэтому все работает, но на грани фола. Очевидно, MS полагает, что баг с разными кучами должен быть обнаружен и исправлен еще на этапе отладки.

В результате дальнейших экспериментов мы пришли к следующему. В описании вектора в MSDN есть любопытная строка: «Note that allocator is not copied when the object is assigned». Если мы правильно поняли смысл этой фразы, то из нее вытекает существование где-то глубоко в недрах STL некоего недокументированного механизма, обеспечивающего передачу **std::allocator** по ссылке внутри STL при конструировании ее объектов (то самое, что мы стремились безуспешно воспроизвести). Наоборот, при выдаче аллокатора во внешний код функцией **get_allocator** (равно как и при приеме аллокатора извне) выполняется копирование. Технически это возможно, поскольку разработчики STL представляют себе внутреннюю реализацию библиотеки и могут применить приемы оптимизации, не доступные сторонним разработчикам.

Таким образом, если эти слова справедливы, значит, при использовании DLL-версии (т. е. динамической) RTL все проблемы должны исчезнуть. Мы изменили соответствующие настройки проекта и убедились в этом.

На основании проведенного исследования выяснилось следующее.

1. При использовании RTL в статической линковке происходит выделение динамически распределяемой памяти из разных куч при помощи дополнительных вызовов **HeapCreate**. При этом DLL использует свою локальную кучу для получения динамической памяти; EXE-модуль выделяет динамическую память из своей кучи. Приведенный пример показывает, что сделано это в целях:

- повышения устойчивости и работоспособности. Как известно, DLL можно использовать в среде, совершенно отличной от той, которая применялась при ее написании. При этом, разумеется, правила выделения и освобождения динамической памяти могут очень сильно различаться. Например, RTL C++ предполагает дополнительное резервирование памяти для сохранения информации о количестве выделенных блоков. RTL Delphi также сохраняет дополнительную информацию, но при этом способы ее хранения определяются уже компанией Borland, а не Microsoft. Затем эта информация используется на этапе освобождения занятой памяти;
- повышения безопасности: проблемы с динамической памятью целиком локализуются в одном модуле — в том, который занимался выделением этой памяти.

2. Благодаря использованию RTL в динамической линковке можно избежать дополнительного выделения куч, так как в этом случае все версии объектов (как объектов, расположенных в DLL, так и объектов

EXE-кода) используют одну и ту же кучу. Прежде всего исчезают проблемы с реаллокированием объектов в разных кучах. За исчезновение проблем приходится платить. Во-первых, тем, что приложение будет требовать **MSVCRT.DLL** для нормального запуска. Во-вторых, могут возникать проблемы, связанные с наличием в контексте одного выполняемого процесса нескольких экземпляров CRT (скажем, один из них будет являться статически связанным кодом, а второй — вызываться посредством механизма DLL). Несмотря на то что линкер умудряется предупреждать о таких ситуациях, все равно можно попасть в ловушку. Например, DLL компонуется со статической версией CRT, а EXE-приложение — с динамической. Основные проблемы будут заключаться в недоступности статических данных в одной версии CRT со стороны другой. Кроме того, следует избегать случаев использования debug- и release-версий CRT в одном процессе.

3. При использовании разных куч необходимо строго соблюдать контексты, в которых происходят дополнительные выделения памяти. В debug-версии подобное несоответствие очень легко отлавливается при помощи специальных проверочных блоков. По-видимому, стандарт не налагает определенных требований к объектам, размещаемым в динамической области памяти, следовательно, пользователь вправе толковать эти требования по собственному разумению. В итоге это может привести к написанию опасного кода. Следовательно, необходимо эти ситуации каким-то образом отслеживать и предупреждать об этом.

4. STL (реализация в VC++ 6.0) предполагает использование модели выделения блоков памяти в динамической куче, отличной от той, которая применяется при обращении к операторам **new/delete**. Сделано это, по-видимому, в целях оптимизации — STL использует потоко-незащищенную реализацию операторов выделения памяти. Напротив, использование стандартных **new/delete** приводит к появлению объектов синхронизации доступа к куче, что, в конечном счете, сильно сказывается на производительности.

Список литературы

1. Microsoft Corporation. Microsoft Developer Network (MSDN). 2008.

2. Microsoft Corporation. Visual Basic 6.0. СПб.: БХВ, 1999.

3. Progz.ru. Портал для программистов // Progz.ru. — http://progz.ru/

4. RSDN.ru. RSDN // Russian Developer Network. — http://www.rsdn.ru/

5. *Архангельский А.Я.* Программирование в C++ Builder 6 и 2006. М.: БИНОМ, 2006.

6. *Коплиен Дж.* Программирование на C++. Классика CS. СПб.: Питер, 2005.

7. *Лафоре Р.* Объектно-ориентированное программирование в C++. Классика CS. СПб.: Питер, 2003.

8. *Макдональд М.* Microsoft Visual Basic.NET: рецепты программирования. СПб.: Питер, 2004.

9. *Марко К.* Delphi 2005. Для профессионалов. СПб.: Питер, 2006.

10. *Петцольд Ч.* Программирование для Microsoft Windows на Microsoft Visual Basic .NET. Т. 1–2. М.: Русская редакция, 2003.

11. *Питрек М.* Секреты программирования в Windows 95. К.: Диалектика, 1996.

12. *Рихтер Дж.* Windows для профессионалов. Программирование для Windows 95 и Windows NT 4 на базе Win32 API. М.: Издательский отдел «Русская редакция», 1997.

13. *Рихтер Дж.*, *Назар К.* Windows via C/C++. Программирование на языке Visual C++. СПб.: Питер, 2008.

14. *Роббинс Дж.* Отладка приложений для Microsoft.NET. М.: Русская редакция, 2008.

15. *Руссилович М.*, *Соломон Д.* Внутреннее устройство Microsoft Windows: Windows Server 2003, Windows XP и Windows 2000. Мастер-класс. СПб.: Питер, 2006.

16. *Страуструп Б.* Язык программирования C++. СПб.: Невский диалект, 1999.

17. *Шамис В.* C++ Builder Borland Developer Studio 2006. Для профессионалов. СПб.: Питер, 2007.

18. *Шульман Э.* Неофициальная Windows 95. К.: Диалектика, 1995.