11 Виды приложений и их структура, библиотеки

11.1 Оверлеи

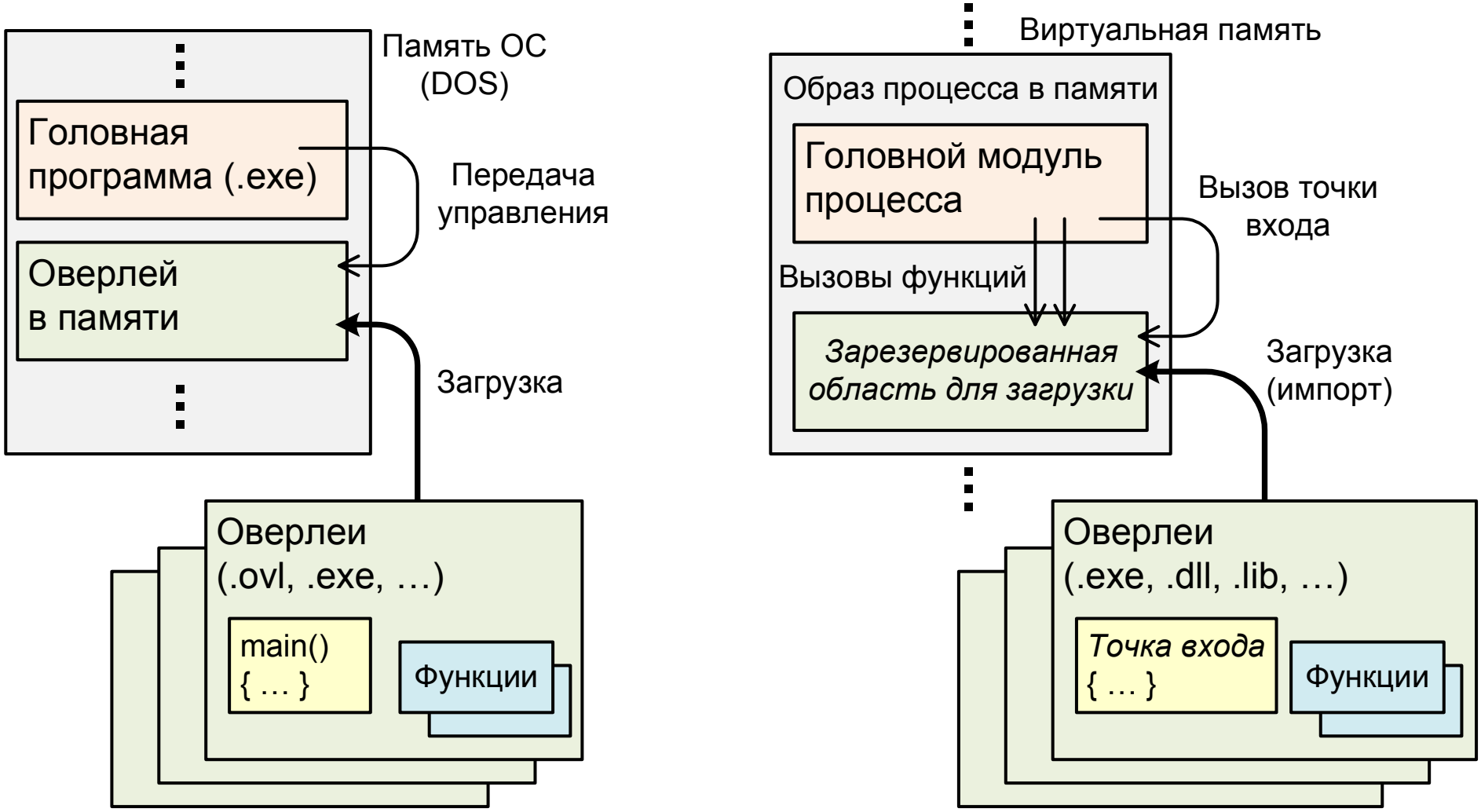
Оверлеи (***overlay***) – программные модули (библиотеки), готовые к загрузке и исполнению, но не самостоятельно, а в составе других программ. Оверлеи не находятся в памяти постоянно, а сменяют (перекрывают) друг друга по мере необходимости.

Использованием оверлеев решается ряд задач:

– ***Экономия*** памяти: загружена только часть кода и данных

– ***Гибкость***: формирование нужной конфигурации программы в ходе ее загрузки

– ***Структурирование*** программы: компоновка ее из готовых модулей, в том числе созданных разными разработчиками и с помощью разных средств программирования


Память ОС
(DOS)
Головная
программа (.exe)
Передача
управления
Оверлей
в памяти
Загрузка
Оверлеи
(.ovl, .exe, …)
main()
{ … }
Функции
Виртуальная память
Образ процесса в памяти
Головной модуль
процесса
Вызов точки
входа
Вызовы функций
Зарезервированная
область для загрузки
Загрузка
(импорт)
Оверлеи
(.exe, .dll, .lib, …)
Точка входа
{ … }
Функции

Оверлеи

Варианты концепции оверлеев реализованы во многих системах: динамические библиотеки DLL в Windows (и OS/2), разделяемые библиотеки в Unix, и т.д.

Эффективное применение возможно при полноценной поддержке со стороны ОС (недостаточность которой в DOS ограничивало использование оверлеев, несмотря на остроту проблемы памяти).

11.2 Динамически подключаемые библиотеки (DLL)

11.2.1 Назначение и структура DLL

Динамически подключаемые (***связываемые***) ***библиотеки*** (***Dynamic Link Library***, ***DLL***) являются наряду с .EXE-файлами разновидностью исполняемых модулей в Windows в формате ***Portable Executable*** (***PE***): они имеют в целом аналогичную структуру, но не могут выполняться самостоятельно, а присоединяются к другим исполняемым файлам. Являясь разновидностью оверлеев, служат в целом тем же целям, изначально имеют поддержку системой.

Сама ОС Windows представляет собой в значительной мере набор динамических библиотек, считающихся стандартными.

Каждый модуль DLL включается в адресное пространство основного процесса, при этом каждое новое подключение модуля дает фактически новый его экземпляр, не зависящий от остальных экземпляров того же модуля.
(В Win16 все экземпляры модуля разделяли одно и то же адресное пространство, что усложняло программирование и снижало безопасность системы.)

Загрузка библиотеки опирается на те же механизмы, что и создание нового процесса. Ввиду того, что сегмент кода библиотеки может разделяться всеми его экземплярами, а вся библиотека разделяется многими процессами и может загружаться ими многократно, для каждой библиотеки, загружаемой каждым процессом, создается счетчик привязок (загрузок), увеличивающийся при загрузке ее данным процессом и уменьшающийся при выгрузке. После обнуления счетчика библиотека удаляется из адресного пространства процесса.

`.EXE`-файлы также могут загружаться аналогично, в адресном пространстве существующего процесса вместо порождения нового.

Содержимое библиотеки (переменные, метки, функции, классы) недоступно (точнее, не известно) вне ее, за исключением специальным образом оформленных ***экспортируемых символов***. Экспортируют в первую очередь функции и методы – API библиотеки.

Экспорт непосредственно данных (переменных) возможен и применяется, но чаще считается нежелательным.

Кроме символов, в библиотеке могут содержаться также ресурсы.

Информация об экспортируемых символах помещается в ***таблицу экспортируемых символов*** модуля, откуда она считывается при подключении библиотеки в процессе ***импорта*** – получении точек доступа к символам.

Экспортируемый символ идентифицируется его ***именем*** и ***индексом*** – порядковым номером в таблице. Имена в таблице экспортируемых символов не обязательно совпадают с их «внутренними» именами – компиляторы позволяют явно задавать «внешние» имена (см. ниже). Индексы символов не обязательно идут последовательно, т.к. также могут указываться явно.

11.2.2 Подключение (импорт) DLL

– ***Явный импорт*** (динамическое подключение) – в любой момент в процессе выполнения программы путем обращения к соответствующим системным вызовам

– ***Неявный импорт*** (статическое подключение) – загрузку всех модулей и импорт символов осуществляет загрузчик до передачи управления «прикладной» точке входа, необходимая для этого информация включается в заголовок исполняемого файла компоновщиком (linker-ом).

Явный импорт обеспечивает наибольшую гибкость и управляемость, возможность обработки ошибок, позволяет подстраиваться под любые соглашения об именах и форматах, но более трудоемок. Неявный импорт происходит прозрачно для программы, но требует соблюдения единых правил. Стандартные библиотеки, системные DLL подключаются неявно.

Явный импорт

Загрузка указанного DLL и включение его в адресное пространство процесса:

```
HINSTANCE LoadLibrary(LPCTSTR lpModuleName);
```

Получение описателя (Handle) уже загруженного модуля:

```
HMODULE GetModuleHandle(LPCTSTR lpModuleName);
```

Декремент счетчика загрузок библиотеки, после обнуления освобождение экземпляра библиотеки и исключение его из адресного пространства процесса:

```
BOOL FreeLibrary(HMODULE hModule);
```

Получение адреса функции или другого символа, импортируемого из библиотечного модуля, по имени или индексу:
– указатель на строку имени: ***lpProc*** **> 0xffff**
– индекс: ***lpProc*** **<= 0xffff**

```
FARPROC GetProcAddress(
  HMODULE hModule, LPCTSTR lpProc);
```

Возвращаемое значение – указатель на функцию вида **(__stdcall *)(void)**; при экспорте символов другого типа требуется соответствующее его приведение.

Головная программа должна сохранить адреса импортированных символов в соответствующих переменных-указателях, возможно с приведением типа, и в дальнейшем использовать их при обращениях к объектам в DLL.

Неявный импорт

Для головной программы достаточно (в оптимистичном случае) включить декларации импортируемых символов при компиляции, обычно посредством заголовочных файлов (`*.h`), и сгенерированные вместе с DLL «оберточные» библиотеки (`*.lib`) при компоновке.

После успешного завершения процедуры загрузки символы из библиотеки становятся доступными для головного модуля. В случае ошибки (обычно отсутствие необходимого модуля) процесс не создается, как правило, с выдачей сообщения о системной ошибке.

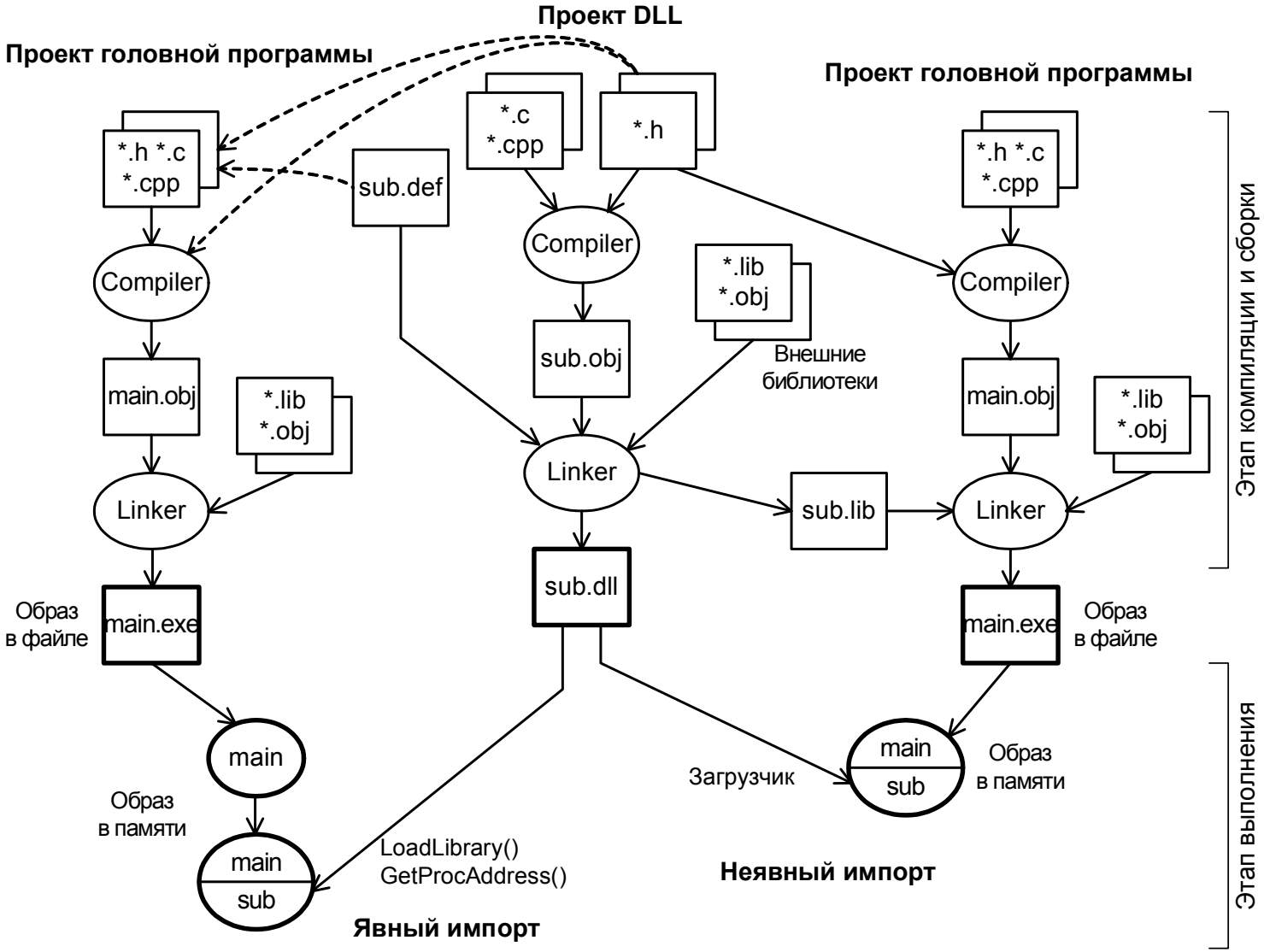
Важное обстоятельство – согласование формата вызова функций: в реализации библиотеки и в использующей ее программе.

Формат точки входа DLL зафиксирован, т.к. к ней обращается операционная система (аналогично точке входа исполняемой программы).

Модификатор **`__stdcall`** – стандартный для Win API формат:
– передача аргументов – через стек
– порядок аргументов – прямой (PASCAL)
– очистка стека – вызванная подпрограмма
– имена – регистро-чувствительные

Для прочих экспортируемых функций формат **`__stdcall`** также обычно предпочтителен, но это не обязательно.

Экспорт и импорт классов обычно затруднен: требуется согласования их описаний и/или явного импорта всех методов. Данные неспецифических типов проблем обычно не вызывают.



Динамически подключаемая библиотека (DLL)

11.2.3 Программирование DLL

Для сред программирования – обычно отдельный тип проекта.

Независимо от способа подключения, библиотека имеет точку входа, описываемую функцией:

```
BOOL APIENTRY DllMain(
  HANDLE hModule, DWORD fCall,
  LPVOID lpReserved);
```

Тип возвращаемого значения **BOOL** совместим со стандартным **int**, ненулевое значение (**TRUE**) соответствует успешному выполнению, нулевое (**FALSE**) – неуспешному, в этом случае действие, инициировавшее обращение к точке входа, завершится с ошибкой, если результат для него критичен.

Модификатор **APIENTRY** стандартный для Win API формат, эквивалентно **__stdcall**.

hModule – описатель (handle) библиотечного модуля (аналогично hInstance для WinMain).

fCall – указывает на тип (причину) обращения к точке входа:
– **DLL_PROCESS_ATTACH** – первичное включение модуля в адресное пространство процесса, модуль может создать необходимые локальные структуры данных (в Win 16 это было практически необходимо)
– **DLL_THREAD_ATTACH** – создание головным процессом нового потока
– **DLL_THREAD_DETACH** – завершение одного из потоков
– **DLL_PROCESS_DETACH** – выгрузка модуля из адресного пространства процесса

Другие среды программирования могут реализовывать другие подходы. Например:
– функция **`DllEntryPoint`**, формат и действие аналогичны (C++ Builder)
– предопределенные секции **`initialization`** и **`finalization`** (Delphi)

Собранный DLL готов для ***явного*** импорта, но от вызывающей программы потребуется знание имен символов для импорта. Обычно это решается предоставлением соответствующего заголовочного (**`*.h`**) файла и (в оптимистичном случае) документированием.

Для ***неявного*** импорта необходимо обеспечить соблюдение единых правил оформления функций, участвующих в интерфейсе, и включение в заголовок **`.exe`**-файла головной программы необходимой информации. Это зависит от среды программирования.

Например, в Visual C++:

В проекте библиотеки экспортируемые символы объявляются с модификатором **`__declspec(dllexport)`**.

В проекте головной программы импортируемые символы объявляются с модификатором **`__declspec(dllimport)`**, кроме того, в этот проект должен быть включен **`.lib`**-файл, сгенерированный вместе с библиотечным модулем.

Для удобства (выполняется автоматически при создании проекта) модификаторы оформляются в макроподстановки внутри директив условной компиляции в заголовочном (`*.h`) файле, который включается в оба проекта (библиотека и головная программа). В результате `.h`-файл содержит информацию для компилятора, а `.lib`-файл – для компоновщика головной программы (подобно обычному `.obj`), и их ошибки будут проявляться на соответствующих стадиях.

```
#ifdef DLL_SUB_EXPORTS
#define DLL_SUB_API __declspec(dllexport)
#else
#define DLL_SUB_API __declspec(dllimport)
#endif

extern DLL_SUB_API char szMyDllData[];

DLL_SUB_API int MyDllFunction( ... );

class DLL_SUB_API CMyDllClass { ... };
```

Проблема соглашений об именах (актуально при любом способе импорта): по умолчанию экспортируются «внутренние» имена символов (сигнатуры), не совпадающие с «видимыми» именами, что осложняет связывание разнородных проектов. При использовании «родственных» наборов инструментов проблема может быть решена опциями и директивами. Радикальное решение – явное стандартное описание символов для экспорта.

В C-ориентированных средствах – традиционно **`.def`**-файл. Секция **`EXPORTS`** описывает все экспортируемые имена. Здесь описана функция **`MyDllFunction()`**, экспортируемая под этим именем и под индексом **`2`**:

```
EXPORTS
MyDllFunction @2
```

Прототип этой функции в исходном тексте:

```
int __stdcall MyDllFunction( ... )
```

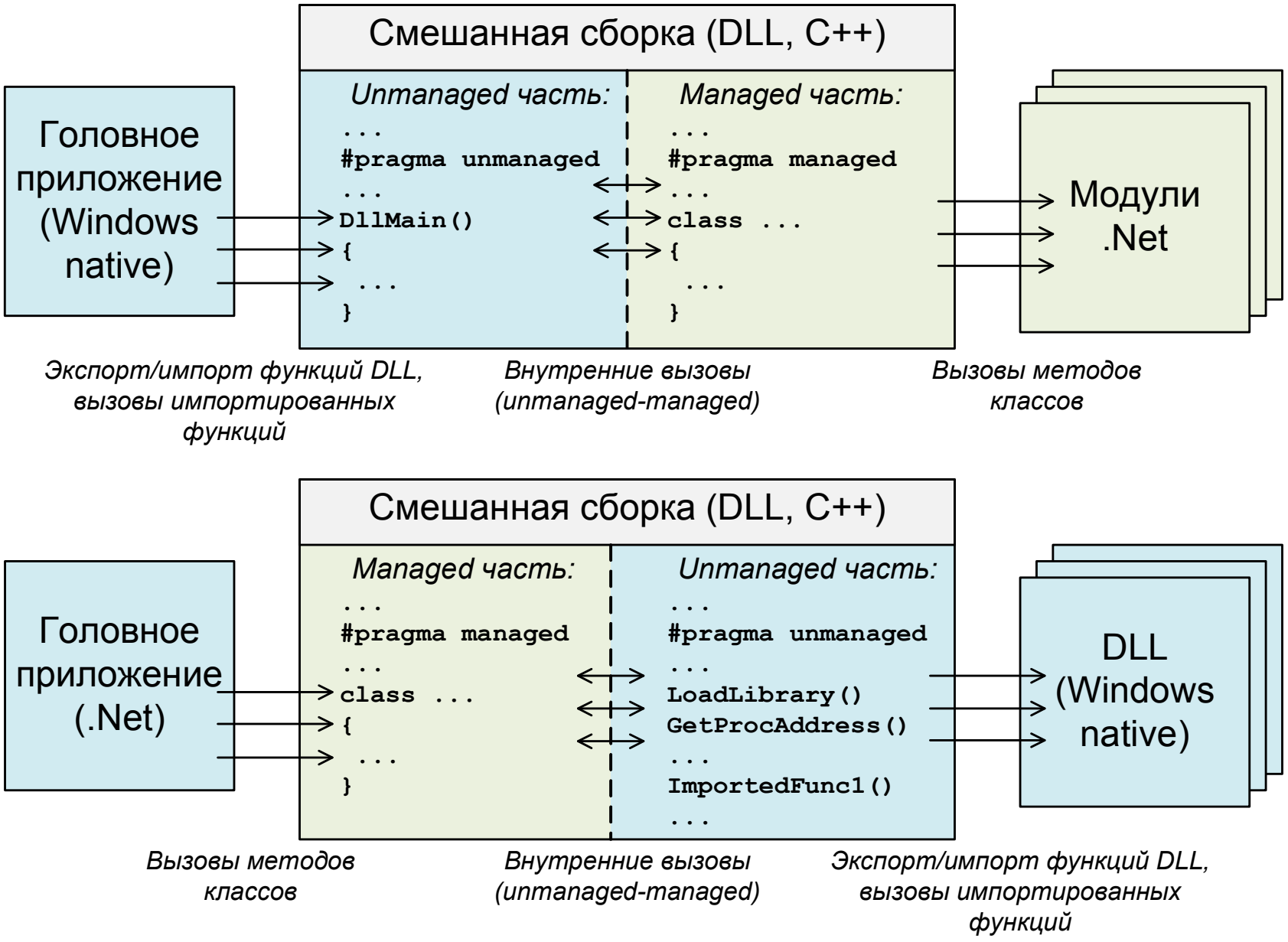
11.3 Смешанные сборки (Mixed Assembly)

Основная рекомендованная технология, позволяющая совместить в одном проекте модули .NET и «обычные» (Native Win API). Это может быть актуально, например, в случаях:

– модификация и сопровождение продуктов, содержащих «старые» модули (приложения **`.exe`** или библиотеки **`.dll`**) и требующих дополнения их .NET кодом

– решение в рамках .NET специфических задач, требующих обращения к Win API и unmanaged кода

В рамках MS Visual Studio – только проекты на C++ (поддержка managed и unmanaged версий одновременно).


Смешанная сборка (DLL, C++)
Unmanaged часть:
...
#pragma unmanaged
...
DllMain()
{
...
}
Managed часть:
...
#pragma managed
...
class ...
{
...
}
Головное приложение (Windows native)
Модули .Net
Экспорт/импорт функций DLL, вызовы импортированных функций
Внутренние вызовы (unmanaged-managed)
Вызовы методов классов
Смешанная сборка (DLL, C++)
Managed часть:
...
#pragma managed
...
class ...
{
...
}
Unmanaged часть:
...
#pragma unmanaged
...
LoadLibrary()
GetProcAddress()
...
ImportedFunc1()
...
Головное приложение (.Net)
DLL (Windows native)
Вызовы методов классов
Внутренние вызовы (unmanaged-managed)
Экспорт/импорт функций DLL, вызовы импортированных функций

Пример: смешанная сборка-DLL для подключения к приложениям Windows Native

Головной модуль – Windows Native (DLL):

```
#include <windows.h>
#include <stdio.h>
#include "MyManagedInterface.h"

#pragma unmanaged

MyManagedInterface mi;

BOOL APIENTRY DllMain( … )
{
  …
  mi.MyInterfaceMethod_1( … );
  mi.MyInterfaceMethod_2( … );
  …
}
```

Подключаемый модуль – managed (MyManagedInterface):

```
#pragma managed
class MyManagedInterface ( … )
{
  …
  MyInterfaceMethod( … ) { … };
  MyInterfaceMethod( … ) { … };
  …
}
```

Точкой входа в unmanaged модуль может быть также WinMain – будет сгенерирован «смешанный» исполняемый файл (**`.exe`**), который будет выполняться самостоятельно.

Вызовы между managed и unmanaged кодом сопровождается ***маршаллингом*** параметров. В вызовах managed методов можно использовать unmanaged указатели.

Требуется использование ключевых слов – модификаторов параметров **`ref`**, **`out`**.

Второй проиллюстрированный вариант менее актуален, так как задача простого импорта функций DLL (в том числе системных) и вызова их из managed-кода решается штатными средствами:

```
[DllImport (”MyDll.dll”)]
static extern … MyDllFunction( … )
```