1. Что такое библиотека?

Обычно простые программы состоят из одного исходного файла. Если программа становится большой, проблема:

* файл, становясь большим, увеличивает время компиляции, и малейшие изменения в исходном тексте автоматически вынуждают тратить время программиста на перекомпиляцию программы
* если над программой работает много человек, то практически невозможно отследить сделанные изменения
* процесс правки и само ориентирование при большом исходном тексте становится сложным, и поиск небольшой ошибки может повлечь за собой вынужденное «изучение» кода заново

Изображение выглядит как снимок экрана, текст, дизайн

Контент, сгенерированный ИИ, может содержать ошибки.

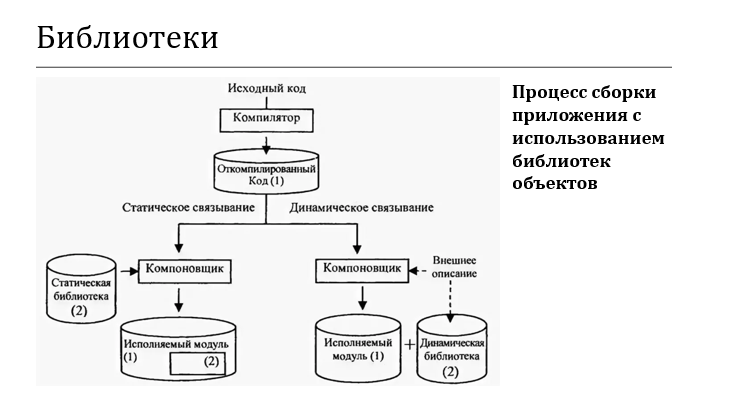
Некоторые из исходных файлов можно было бы использовать в нескольких программах. Первым делом эти файлы можно скомпилировать только один раз и затем уже по необходимости компоновать их с разными исполняемыми файлами. И хотя такой подход уменьшает время компиляции, он все равно не избавляет от необходимости каждый раз указывать все obj файлы на этапе компоновки. Более того, с увеличением количества таких файлов можно создать неразбериху в каталоге проекта. Чтобы обойти это, создаем библиотеку:

**Библиотека объектных файлов** – это файл, содержащий несколько obj файлов, которые будут использоваться вместе на стадии сборки (**связывания**, линковки) программы. Нормальная библиотека содержит **символьный индекс**, состоящий из названий функций, переменных и т. д., которые содержатся в библиотеке. Это позволяет ускорить процесс сборки программы

Библиотеки объектов бывают двух видов: **статические и динамические (разделяемые)**

**Связывание** - компоновка программы из многих obj модулей. Поскольку каждый из obj модулей в составе программы был получен в результате отдельного процесса трансляции, который работает только с одним конкретным модулем, обращения к процедурам и данным, расположенным в других модулях, в obj модулях не содержат актуальных адресов. Загрузчик «видит» все obj модули, входящие в состав программы, и он может вставить в обращения к внешним точкам правильные адреса

Когда программа компилируется, каждый её obj модуль создаётся отдельно, и на этом этапе компилятор не может знать, где находятся данные или функции из других модулей. Он лишь ставит заглушки — адреса или ссылки, которые будут заменены в процессе связывания.



**Виды связывания** - раннее(во время трансляции или во время сборки) и позднее(при загрузке или отложенное[итеративное(явная загрузка), декларативное(неявная загрузка)]).

1. Что такое статическая библиотека?

**Статическая библиотека** - обычный файл, содержащим копии всех помещенных в него obj файлов. Это архив, там хранятся атрибуты для каждого obj файла: права доступа, id пользователя и группы и время последнего изменения. В Linux название - **libname.a, в win пофик. Статическая библиотека содержит lib файл (obj)**

+ набор часто используемых obj файлов можно поместить в единую библиотеку, при этом не нужно будет перекомпилировать ориг код.  
+ упрощение команды для компоновки. не перечисляем список obj файлов, а указываем только имя библиотеки. Компоновщик знает, как выполнять поиск по имени и извлекать объекты, необходимые для создания исполняемого файла.   
может быть несколько копий одних и тех же obj модулей, поэтому минусы:   
-дисковое пространство хранит одни и те же obj модули  
- виртуальная память потребляется много, если несколько программ одноврем.  
- если obj модуль требует изменение, то заново компонуем все исполняемые файлы, где используется модуль

**Раннее (статическое) связывание.** На этапе компоновки или трансляции obj-код содержащийся в библиотеке внедряется в будущий исполняемый файл.

У нас использовалось на этапе трансляции. Есть еще во время сборки - используется pragma.

1. Как подключить статическую библиотеку?

С помощью флага -l. Далее рассказать про cmake и в консоли.

Рассматриваемый далее пример для Windows будет собираться идентичным образом и на ОС Linux (утилита [**llvm-ar**](https://llvm.org/docs/CommandGuide/llvm-ar.html) может отсутствовать, что решается либо установкой через **sudo apt install llvm**, либо использованием утилиты [**ar**](https://pubs.opengroup.org/onlinepubs/9799919799/utilities/ar.html))

Этот пример не будет корректно работать на Linux из-за вывода char и wchar\_t друг за другом.

Изображение выглядит как текст, снимок экрана, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.

Для создания и редактирования статических библиотек используется утилита **ar** или [**llvm-ar**](https://llvm.org/docs/CommandGuide/llvm-ar.html): **ar <operation>[modifiers] <archive> [files]**

Аргумент **operation** состоит из буквы, которая является кодом операции, а **modifiers** - модификаторами, влияющим на то, как эта операция будет выполняться, которые записываются как набор букв слитно с буквой операции

Аргумент **archive** представляет собой имя создаваемой библиотеки, а **files** – obj файлы, из которых она будет состоять.

Среди часто используемых операций и модификаторов можно выделить:

* **r**.Вставляет obj файл в архив, заменяя им любой существующий файл с тем же именем. Это стандартный способ создания и обновления архивов. Используя с модификатором **c,** заставляет создавать библиотеку, если ее нет.
* **t**. Выводит оглавление архива. По умолчанию выводятся только имена obj файлов. Но если доп указать **v**, можно просмотреть все атрибуты каждого файла в архив.

Изображение выглядит как текст, снимок экрана, программное обеспечение, число

Контент, сгенерированный ИИ, может содержать ошибки.

* **d**. Удаляет из архива заданный модуль.

Кроме этого, стоит отметить, что в примере была использована утилита [**llvm-ranlib**](https://llvm.org/docs/CommandGuide/llvm-ranlib.html)(вне llvm существует её полный аналог **ranlib**)

Данная утилита позволяет добавить к нашей библиотеке (по факту до выполнения runlib, наш файл является не более чем просто архивом) индекс символов, т. е. список вложенных в библиотеку функций и переменных

В рамках примера утилита **llvm-ar** (и **ar**) проводит индексацию по умолчанию и явно вызывать **llvm-ranlib** не имеет смысла. Но стоит

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.

Изображение выглядит как текст, программное обеспечение, Мультимедийное программное обеспечение, Значок на компьютере

Контент, сгенерированный ИИ, может содержать ошибки.Изображение выглядит как текст, Шрифт, программное обеспечение, линия

Контент, сгенерированный ИИ, может содержать ошибки.

Изображение выглядит как текст, снимок экрана, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

[**include\_directories**](https://cmake.org/cmake/help/latest/command/include_directories.html) – указывает на каталоги, которые будут использоваться для поиска заголовочных файлов (include)

[**add\_library**](https://cmake.org/cmake/help/latest/command/add_library.html) – необходимо собрать библиотеку из указанных файлов (в данном случае статическую поскольку указан параметр STATIC) (.c)

[**target\_link\_libraries**](https://cmake.org/cmake/help/latest/command/target_link_libraries.html) – для связывания приложения с библиотекой. Также через неё можно передавать и другие параметры компоновщику

Cборка примера на Linux без CMake отличия: название libtest.**a** вместо libtest.**lib**, значение опции target **x86\_64-pc-linux** вместо **x86\_64-pc-win32**

Также стоит отметить, что очень важен порядок, в котором передаются входные файлы драйверу компилятора (clang не компилятор, а программа, которая вызывает всё что необходимо для сборки приложения):

СНАЧАЛА ФАЙЛЫ ПРИЛОЖЕНИЯ, ЗАТЕМ БИБЛИОТЕКИ! ДАННОЕ ПРАВИЛО ИСХОДИТ ИЗ НЕОБХОДИМОСТИ СНАЧАЛА ПЕРЕДАТЬ ТОТ КОД, В КОТОРОМ ИСПОЛЬЗУЕТСЯ ФУНКЦИЯ, А ТОЛЬКО ПОТОМ ТОТ В КОТОРОМ ОНА ОБЪЯВЛЕНА!

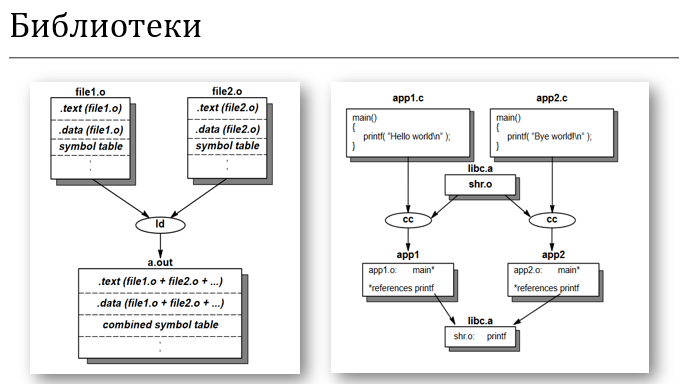
В примере выше раннее связывание **во время сборки (link-time)**. Компилятор MSVC поддерживает связывание во время трансляции через директиву препроцессора [**pragma**](https://learn.microsoft.com/en-us/cpp/preprocessor/comment-c-cpp?view=msvc-170):

Изображение выглядит как Шрифт, снимок экрана, текст, Графика

Контент, сгенерированный ИИ, может содержать ошибки.

В случае использования данной директивы рассмотренный ранее пример может быть собран следующей командой: (здесь не нужна .lib, он используется в pragma)





1. На каком этапе сборки происходит непосредственно загрузка кода?
2. Что такое динамическая библиотека?

**Разделяемые** (**динамические**) **библиотеки (в Win Dynamic Link Library, в Linux разделяемые объекты shared objects)**: одна копия obj модуля разделяется между всеми программами, задействующими его. Obj модули **не копируются** в компонуемый исполняемый файл; вместо этого **единая копия библиотеки** загружается в память при запуске первой программы, которой требуются ее obj модули. Если позже будут запущены другие программы, использующие эту разделяемую библиотеку, они обращаются к копии, уже загруженной в память.

+можно загружать в адресное пространство динамически, что позволяет приложению подгружать нужный код.  
+ использование разных ЯП (приложение c#, логика с++)  
+ проще управлять проектом  
+ экономия памяти (исполняемые файлы требуют меньше места на диске и в виртуальной памяти (при выполнении))  
+ разделение ресурсов

+ упрощение локализации

DLL-файл представляет собой файл в формате **Portable Executable** ([**PE**](https://learn.microsoft.com/en-us/windows/win32/debug/pe-format))

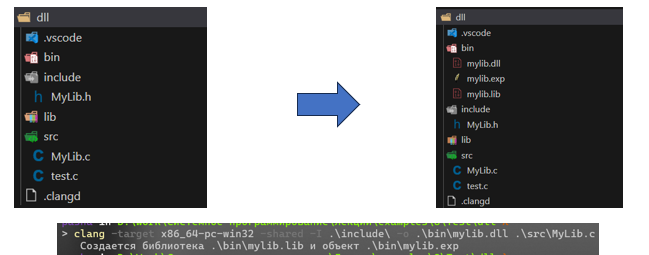
SO-файл представляет собой файл в формате **Executable and Linkable Format** ([**ELF**](https://gist.github.com/x0nu11byt3/bcb35c3de461e5fb66173071a2379779))

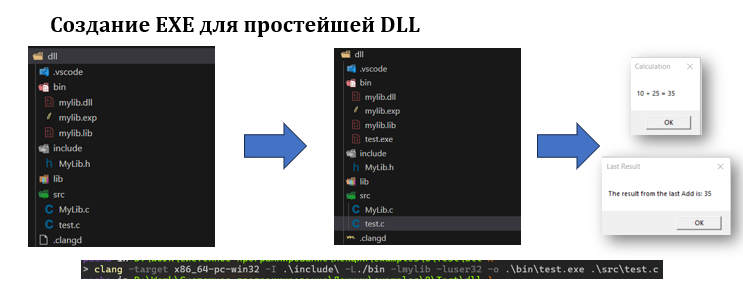
Чтобы приложение (или другая DLL) могло вызывать функции, содержащиеся в DLL, образ ее файла нужно сначала спроецировать на адресное пространство вызывающего процесса. Это достигается либо за счет неявного связывания при загрузке, либо за счет явного – в период выполнения.

Как только DLL спроецирована на адресное пространство вызывающего процесса, ее функции доступны всем потокам этого процесса. Фактически библиотеки при этом теряют почти всю индивидуальность: для потоков код и данные DLL – просто доп код и данные, оказавшиеся в адресном пространстве процесса. Когда поток вызывает из DLL какую-то функцию, та считывает свои параметры из стека потока и размещает в этом стеке собственные локальные переменные. Кроме того, любые созданные кодом DLL объекты принадлежат вызывающему потоку или процессу – DLL ничем не владеет!

в конечный DLL:  
Библиотека импорта - LIB-файл, нужен для любого exe-модуля, содержит идентификаторы.   
Раздел экспорта - таблица экспортируемых идентификаторов.  
Относительный виртуальный адрес каждого идентификатора. 







Стоит обратить внимание на тот факт, что в примере файл mylib.dll располагается в том же каталоге, что и исполняемый файл. Это связано с тем, где система будет искать библиотеку. Если для используемой библиотеки используется только её имя (нет полного пути), то поиск DLL осуществляется в следующей последовательности:

* Каталог с EXE
* Текущий каталог процесса
* Системный каталог Windows
* Основной каталог Windows
* Каталоги PATH

Как можно заметить в коде и заголовочном файле библиотеки применяются модификаторы:

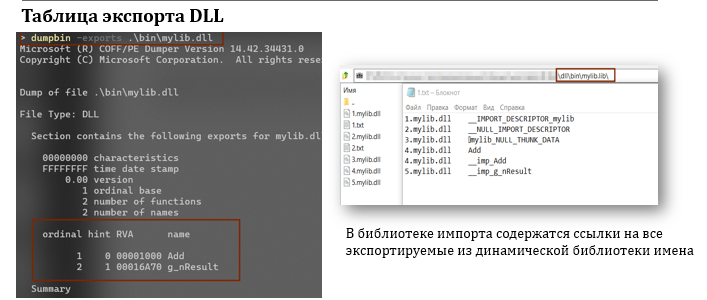
* [**\_\_declspec(dllimport)**](https://learn.microsoft.com/en-us/cpp/cpp/dllexport-dllimport?view=msvc-170)– такой модификатор означает, что данная переменная или функция импортируются из DLL (указывается с исполняемых модулях или библиотеках, которые зависят от других библиотек)
* [**\_\_declspec(dllexport)**](https://learn.microsoft.com/en-us/cpp/cpp/dllexport-dllimport?view=msvc-170)– такой модификатор означает, что данная переменная или функция экспортируется из DLL (указывается в самой библиотеке)

**\_\_declspec(dllexport)**

Если он указан перед переменной, прототипом функции или C++-классом, компилятор С/С++ встраивает в конечный OBJ-файл дополнительную информацию. Она понадобится компоновщику при сборке DLL из OBJ-файлов.

Обнаружив такую информацию, компоновщик создает LIB-файл со списком идентификаторов, экспортируемых из DLL. Этот LIB-файл нужен при сборке любого EXE-модуля, ссылающегося на такие идентификаторы и называется **библиотекой импорта**

Компоновщик также вставляет в конечный DLL-файл **таблицу** экспортируемых идентификаторов – [**раздел экспорта**](https://learn.microsoft.com/en-us/windows/win32/debug/pe-format), в котором содержится список (в алфавитном порядке) идентификаторов экспортируемых функций, переменных и классов. Туда же помещается [**относительный виртуальный адрес**](https://learn.microsoft.com/en-us/windows/win32/debug/pe-format)(**relative virtual address, RVA**) каждого идентификатора внутри DLL-модуля



Как можно заметить из таблицы экспорта, у каждой функции есть **порядковый номер** ([**ordinal**](https://learn.microsoft.com/en-us/cpp/build/exporting-functions-from-a-dll-by-ordinal-rather-than-by-name?view=msvc-170)) и **имя** (**name**)

Соответственно существуют два способа экспорта функций по номеру и по имени

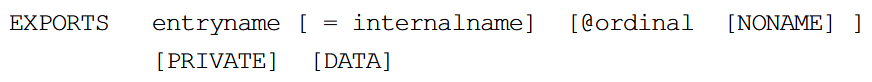
Чтобы экспортировать по имени достаточно применить модификатор **\_\_declspec(dllexport)**

Изображение выглядит как текст, Шрифт, снимок экрана, число

Контент, сгенерированный ИИ, может содержать ошибки.



def-файл (файл определений)  
+ настраивать экспорт по порядковому номеру  
+ указать конечное имя для функции  
+ область видимости экспортируемых функций  
+ можно и без имен, чисто по порядковым номерам



Параметр ***entryname (один обязательный)*** - имя экспортируемой функции или переменной. Если это имя отличается от внутреннего имени функции или переменной в DLL, то это внутреннее имя задается параметром ***internalname***.

Параметр **@*ordinal*** задает порядковый номер экспортируемого имени

Атрибут ***DATA*** - экспортируемое имя является именем переменной

Ключевое слово ***PRIVATE*** запрещает размещение экспортируемого имени в библиотеке импорта.

Атрибут ***NONAME*** указывает на то, что в таблице экспорта не будет храниться данное экспортируемое имя. В этом случае импорт адреса возможен только по его порядковому номеру

Изображение выглядит как текст, снимок экрана, Шрифт, число

Контент, сгенерированный ИИ, может содержать ошибки. Изображение выглядит как текст, снимок экрана, Шрифт, дизайн

Контент, сгенерированный ИИ, может содержать ошибки.

Эти файлы удобно использовать тогда, когда DLL содержит большое число экспортируемых функций и переменных. В этом случае поиск экспортируемых имен занимает очень много времени. Поэтому для более быстрой работы приложения функции и переменные импортируются по их порядковым номерам. Полезен при неявном связывании, когда весь импорт системой производится по номерам и в именах нету смысла, а их отсутствие может сэкономить ресурсы.

Если мы говорим о создании библиотек на языках отличных от С, то скорее всего при создании символов для таблицы экспорта/импорта компилятор преобразует имена из кода в понятные для себя имена. Данный процесс называется **name decoration** или **name mangling** (компилятор C такого не делает, он сохраняет имя функций)

.def-файл позволяет оставить имена функций в том виде, который понятен человеку

Альтернативным способом сохранения имени функции будет модификатор **extern “C”**

  
Изображение выглядит как текст, снимок экрана, меню, дизайн

Контент, сгенерированный ИИ, может содержать ошибки.

**\_\_declspec(dllimport)**

Импортируя идентификатор, необязательно прибегать к \_\_declspec(dllimport) – можно использовать стандартное ключевое слово extern языка C. Но компилятор создаст чуть более эффективный код, если ему будет заранее известно, что идентификатор, на который мы ссылаемся, импортируется из LIB-файла DLL-модуля

Разрешая ссылки на импортируемые идентификаторы, компоновщик создает в конечном EXE-модуле [**раздел или таблицу импорта**](https://learn.microsoft.com/en-us/windows/win32/debug/pe-format) (**imports section**). В нем перечисляются DLL, необходимые этому модулю, и идентификаторы, на которые есть ссылки из всех используемых DLL

Изображение выглядит как текст, снимок экрана, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

Изображение выглядит как текст, снимок экрана, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

**Общая картина работы с DLL (неявный способ)**

Изображение выглядит как текст, Шрифт, снимок экрана

Контент, сгенерированный ИИ, может содержать ошибки.  
Изображение выглядит как текст, снимок экрана, диаграмма, линия

Контент, сгенерированный ИИ, может содержать ошибки.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.

Изображение выглядит как текст, снимок экрана, Мультимедийное программное обеспечение, программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.

Попытка запуска провалилась, хоть и файл разделяемой библиотеки находится в той же папке. Связано это с тем, где Linux ищет файлы библиотек, а именно: динамический компоновщик анализирует список динамических зависимостей программы и находит соответствующие библиотечные файлы, используя набор заранее заданных правил. Часть этих правил основывается на списке стандартных каталогов, в которых обычно хранятся разделяемые библиотеки (к примеру, /lib и /usr/lib). Причина сообщения об ошибке, приведенного ранее, заключается в том, что библиотека находится в текущем каталоге, которая не учитывается динамическим компоновщиком при выполнении поиска.

Для оповещения динамического компоновщика о том, что разделяемая библиотека находится в нестандартном месте, можно воспользоваться переменной среды **LD\_LIBRARY\_PATH**, указав соответствующий каталог в качестве одного из элементов списка, разделенного двоеточиями. После указания текущего каталога приложение успешно запустилось.

**Таблица импорта программы**

Изображение выглядит как текст, Шрифт, Мультимедийное программное обеспечение, программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.

Как можно заметить в отличие от прошлого варианта изменилось следующее:

* Подключать заголовочный файл не требуется
* Появились вызовы функций API для работы с DLL
* При сборке приложения больше не требуется указывать использование DLL

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.

**2 СПОСОБ** (ЯВНАЯ ЗАГРУЗКА DLL в период выполнения приложения))

Поток приложения явно загружает DLL в адресное пространство процесса, получает виртуальный адрес необходимой DLL-функции и вызывает ее по этому адресу.

+ подключать .h не требуется

- появились вызовы функций API для работы с DLL  
+ при сборке приложения не требуется указывать DLL

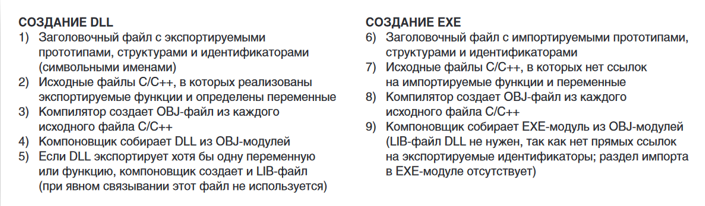
Изображение выглядит как текст, снимок экрана, программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.



Таблица импорта. Если вызвать утилиту dumpbin, то для текущего примера мы не найдём в таблице импорта нашей библиотеки, ей не откуда там взяться, компоновщик ничего не знает о библиотеке в данном случае.

**Общая картина работы с DLL (явный способ)**



Изображение выглядит как текст, снимок экрана, диаграмма, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

**Явная загрузка SO-файла**

* Появились вызовы функций API для работы с SO
* При сборке приложения больше не требуется указывать использование библиотеки

test.c

Изображение выглядит как текст, снимок экрана, программное обеспечение, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.



**Функции WinAPI для работы с DLL**

В любой момент поток может спроецировать DLL на адресное пространство процесса, вызвав одну из двух функций:

Изображение выглядит как текст, Шрифт, снимок экрана, белый

Контент, сгенерированный ИИ, может содержать ошибки.

Обе функции ищут образ DLL-файла и пытаются спроецировать его на адресное пространство вызывающего процесса.

Если необходимость в DLL отпадает, ее можно выгрузить из адресного пространства процесса, вызвав функцию:



Поток получает адрес экспортируемого идентификатора из явно загруженной DLL вызовом **GetProcAddress**:

Изображение выглядит как текст, Шрифт, белый, типография

Контент, сгенерированный ИИ, может содержать ошибки.

Заметьте: тип параметра ***pszSymbolName*** – PCSTR, а не PCTSTR. Это значит, что функция **GetProcAddress** принимает только ANSI-строки – ей нельзя передать Unicode-строку. А причина в том, что идентификаторы функций и переменных в разделе экспорта DLL всегда хранятся как ANSI-строки

Вторая форма параметра ***pszSymbolName*** позволяет указывать порядковый номер нужной функции:



Здесь подразумевается, что Вам известен порядковый номер (2) искомого идентификатора, присвоенный ему автором данной DLL. Microsoft настоятельно не рекомендует пользоваться порядковыми номерами

**Функции POSIX для работы с SO**

Функция [**dlopen**](https://pubs.opengroup.org/onlinepubs/9799919799/functions/dlopen.html) загружает в виртуальное адресное пространство вызывающего процесса разделяемую библиотеку с именем libfilename и инкрементирует счетчик открытых ссылок на нее :

Изображение выглядит как текст, Шрифт, линия, алгебра

Контент, сгенерированный ИИ, может содержать ошибки.

Если разделяемая библиотека, указанная с помощью аргумента libfilename, зависит от других библиотек, то **dlopen** загрузит их автоматически. При необходимости эта процедура выполняется рекурсивно. Мы будем называть набор загруженных таким образом библиотек **деревом зависимостей.**

Функция [**dlclose**](https://pubs.opengroup.org/onlinepubs/9799919799/functions/dlclose.html) закрывает библиотеку:

Изображение выглядит как текст, снимок экрана, Шрифт, линия

Контент, сгенерированный ИИ, может содержать ошибки.

Функция [**dlsym**](https://pubs.opengroup.org/onlinepubs/9799919799/functions/dlsym.html) ищет именованный символ (symbol – функцию или переменную) в библиотеке, на которую указывает дескриптор (handle), и в ее дереве зависимостей.

Изображение выглядит как текст, снимок экрана, Шрифт, линия

Контент, сгенерированный ИИ, может содержать ошибки.

При получении ошибки из **dlopen** или другой функции, входящей в программный интерфейс **dlopen**, можно попытаться узнать ее причину, получив указатель на соответствующую строку с текстом ошибки, используя вызов [**dlerror**](https://pubs.opengroup.org/onlinepubs/9799919799/functions/dlerror.html)**:**

Изображение выглядит как текст, Шрифт, алгебра, снимок экрана

Контент, сгенерированный ИИ, может содержать ошибки.

**Общий алгоритм загрузки и отчистки библиотеки из памяти**

Функцию **dlopen**/**LoadLibrary** можно вызвать несколько раз для одной и той же библиотеки. При этом загрузка будет выполнена лишь при первом вызове, а во всех последующих случаях станет возвращаться одно и то же значение HANDLE

Однако программный интерфейс **dlopen**/**LoadLibrary** хранит счетчик ссылок для каждого дескриптора

С каждым вызовом **dlopen**/**LoadLibrary** он инкрементируется, а декрементация происходит при вызове **dlclose**/**FreeLibrary**; последний выгружает библиотеку из памяти только в том случае, если счетчик равен 0

В DLL может быть указана функция входа/выхода (одна). Система вызывает ее в некоторых ситуациях сугубо в информационных целях, и обычно она используется DLL для инициализации и очистки ресурсов в конкретных процессах или потоках. Если Вашей DLL подобные уведомления не нужны, Вы не обязаны реализовывать эту функцию. Пример – DLL, содержащая только ресурсы

В Linux в SO такой функции быть не может, однако вместо этого можно определить одну или несколько функций, которые будут автоматически вызываться при загрузке и выгрузке разделяемой библиотеки

Изображение выглядит как текст, программное обеспечение, Мультимедийное программное обеспечение, снимок экрана

Контент, сгенерированный ИИ, может содержать ошибки.

неявное -   
.h .c .o .dll - экспорт хотя бы одной функцию и компоновщик создает lib файл  
.h .c .o - учитывает ссылки и lib создает exe

При неявном подключении (implicit linking) линкеру передается библиотека импорта (расширение lib), содержащая список переменных и функций DLL, которые могут использовать приложения. Обнаружив, что программа обращается хотя бы к одной из них, линкер добавляет в целевой exe-файл таблицу импорта. Таблица импорта содержит список всех DLL, которые использует программа, с указанием конкретных переменных и функций, к которым она обращается. Позже, когда exe-файл будет запущен, загрузчик проецирует все DLL, перечисленные в таблице импорта, на адресное пространство процесса;

В случае неявного подключения все библиотеки, используемые приложением, загружаются в момент его запуска и остаются в памяти до его завершения (даже если другие запущенные приложения их не используют). Это может привести к нерациональному расходу памяти, а также заметно увеличить время загрузки приложения, если оно использует очень много различных библиотек. Кроме того, если хотя бы одна из неявно подключаемых библиотек отсутствует, работа приложения будет немедленно завершена. Явный метод лишен этих недостатков, но делает программирование более неудобным, поскольку требуется следить за своевременными вызовами LoadLibrary и соответствующими им вызовами FreeLibrary, а также получать адрес каждой функции через вызов GetProcAddress.

Явное связывание, требует, чтобы в программе содержались конкретные указания относительно того, когда именно необходимо загрузить или освободить библиотеку DLL. Далее программа получает адрес запрошенной точки входа и использует этот адрес в качестве указателя при вызове функции. В вызывающей программе функция не объявляется; вместо этого в качестве указателя на функцию объявляется переменная. Поэтому во время компоновки программы присутствие библиотеки не является обязательным. Для выполнения необходимых операций требуются три функции: LoadLibrary (или LoadLibraryEx), GetProcAddress и FreeLibrary.

Если используемая вами библиотека является статической, компоновщик скопирует объектный код этих функций непосредственно из библиотеки и вставит его в исполняемый файл.  
Если библиотека является динамической, компоновщик не будет вставлять объектный код, а вставит заглушку, которая по сути сообщает, что эта функция находится в этой DLL в этом месте.

Неявное связывание — ОС загружает библиотеку DLL в тот момент, когда она используется EXE. EXE вызывает экспортированные функции DLL так же, как статически скомпонованные и включенные в состав самого EXE функции. Явное связывание — ОС загружает DLL по запросу во время выполнения. EXE должен явно загружать и выгружать ее. Кроме того, в нем должен быть настроен указатель функции для доступа к каждой используемой функции из библиотеки DLL. В отличие от вызовов функций в статически скомпонованной или неявно связанной библиотеке DLL, при работе с явно связанной DLL EXE должен вызывать экспортированные функции с помощью указателей функций.

1. Какой механизм лежит в основе работы динамических библиотек?
2. Назовите два способа подключения динамической библиотеки? Кратко поясните порядок подключений.
3. Что такое библиотека импорта?
4. Для чего нужен extern “C”?
5. Функции жиненного цикла динамических библиотек в Windows и Linux?