Обычно простые программы состоят из одного исходного файла. Если программа становится большой, то при работе с ней может возникнуть несколько достаточно серьезных проблем:

* 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

Поэтому при разработке программ рекомендуется разбивать их на куски, которые функционально ограничены и закончены!

Кроме упомянутых ранее ситуаций, часто бывает так, что некоторые из исходных файлов можно было бы использовать в нескольких программах

Первым делом, чтобы не заниматься лишней работой, 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 объектные файлы на этапе компоновки. Более того, с увеличением количества таких файлов можно создать неразбериху в каталоге проекта

Чтобы обойти эти проблемы, можно сгруппировать набор объектных файлов в единую сущность – **библиотеку объектов** (или **объектную библиотеку**)

**Библиотека объектных файлов** – это файл, содержащий несколько объектных 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)**



**Статическая библиотека**, по сути, является обычным файлом, содержащим копии всех помещенных в него объектных файлов. В архиве также хранятся различные атрибуты для каждого объектного файла, включая права доступа, числовые идентификаторы пользователя и группы и время последнего изменения

В Unix-подобных системах статическим библиотекам принято давать имена вида **libname.a**

В Windows каких-либо общепринятых правил наименования статических библиотек нет

Как понятно из названия при использовании статической библиотеки речь идёт о **раннем (статическом) связывании**. В результате такого связывания весь объектный код содержащийся в библиотеке внедряется в будущий исполняемый файл **на этапе компоновки или трансляции** (последнее – это особый случай о котором чуть позже)

Как уже упоминалось, да и видно на примере, для создания и редактирования статических библиотек используется утилита **ar** или [**llvm-ar**](https://llvm.org/docs/CommandGuide/llvm-ar.html), которая имеет следующую форму:

**ar <operation>[modifiers] <archive> [files]**

**<operation>** — это **основное действие**, которое вы хотите выполнить с архивом. Это обязательная часть.

**[modifiers]** — это **модификаторы**, которые уточняют или изменяют поведение основной операции. Они пишутся сразу после операции, без пробела, и являются *необязательными*.

* **r** (от англ, replace - «заменить»). Вставляет объектный 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 (от англ, create) библиотеку, если ее нет
* **t** (от англ, table of contents - «оглавление»). Выводит оглавление архива. По умолчанию выводятся только имена объектных файлов. Но если дополнительно указать параметр **v** (от англ, verbose - «подробно»), можно просмотреть все атрибуты каждого файла в архив
* **d** (от англ, delete - «удалить»). Удаляет из архива заданный модуль

**llvm-ranlib** 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

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

[**add\_library**](https://cmake.org/cmake/help/latest/command/add_library.html) – команда для указания, что необходимо собрать библиотеку из указанных файлов

* STATIC: Собрать **статическую библиотеку** (.a, .lib). Код библиотеки будет скопирован в исполняемый файл или другую библиотеку при компоновке.
* SHARED: Собрать **динамическую (разделяемую) библиотеку** (.so, .dll). Библиотека останется отдельным файлом и будет загружаться во время выполнения.
* MODULE: Особый тип разделяемой библиотеки, которая не предназначена для прямой компоновки (линковки), а обычно загружается динамически во время выполнения (например, плагины). Похожа на SHARED.
* OBJECT: Создать "объектную библиотеку". Это не настоящая библиотека, а просто группа объектных файлов (.o, .obj), скомпилированных из source1, source2... Эту цель потом можно "прилинковать" к другим целям (статическим, разделяемым, исполняемым), и её объектные файлы будут использованы при их компоновке. Полезно, чтобы избежать дублирования компиляции одних и тех же файлов для разных целей.

[**target\_link\_libraries**](https://cmake.org/cmake/help/latest/command/target_link_libraries.html) – команда для связывания приложения с библиотекой (чаще всего используется для библиотек которые собираются в этом же проекте, но можно и для внешних). Также через неё можно передавать и другие параметры компоновщику

|  |
| --- |
| **Связать программу (цель CMake) с библиотекой** с помощью target\_link\_libraries означает:   * 1. **Указать компоновщику (linker):** "При финальной сборке этой цели (например, my\_app), используй не только её собственные объектные файлы (например, main.o), но и ищи недостающие символы (функции, переменные, которые вызываются в main.o, но не определены там) в указанной библиотеке (например, libmath.a или libmath.so)."   2. **Обеспечить разрешение символов:** Компоновщик просматривает указанную библиотеку и находит в ней реализации символов, помеченных как "неразрешенные" (unresolved) в объектных файлах цели. Он заменяет вызовы этих символов на реальные адреса или механизмы вызова из библиотеки.   3. **Включить код (для статики) или зависимости (для динамики):**      + Для **статической** библиотеки: Код необходимых функций/данных из библиотеки *копируется* в конечный исполняемый файл цели.      + Для **динамической** библиотеки: В конечный файл цели записывается информация о том, что при запуске потребуется загрузить указанную динамическую библиотеку, и как найти в ней нужные символы во время выполнения. |

Также стоит отметить, что очень важен порядок в котором передаются входные файлы драйверу компилятора.

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!

#pragma comment(lib, "library\_name.lib") — это **директива компилятора Microsoft Visual C++ (MSVC)**, которая встраивает инструкцию для **компоновщика (linker)** прямо в объектный файл (.obj).

1. Эта директива обрабатывается **компилятором** (во время фазы компиляции).
2. Компилятор не выполняет само связывание, но он **встраивает специальную запись (инструкцию) прямо в создаваемый объектный файл (.obj)**.
3. Эта запись говорит **компоновщику (linker)**, который будет обрабатывать этот .obj файл позже: "Тебе нужно будет подключить библиотеку library\_name.lib".
4. **Итог – связывание во время трансляции**

Когда программа компонуется со статической библиотекой (или вовсе без использования библиотек), **итоговый исполняемый файл содержит копии всех объектных модулей, скомпонованных с программой**.

Таким образом, несколько разных программ могут содержать в себе копии одних и тех же объектных модулей. Подобная избыточность несет в себе несколько недостатков:

* дисковое пространство уходит на хранение нескольких копий одних и тех же объектных модулей. Такие потери могут быть значительными
* если несколько программ, применяющих одни и те же модули, выполняются одновременно, каждая из них будет хранить в **виртуальной памяти** свою отдельную копию этих модулей, увеличивая тем самым потребление виртуальной памяти в системе
* если объектный модуль статической библиотеки требует каких-либо изменений (возможно, нужно закрыть дыру в безопасности или исправить ошибку), придется заново компоновать все исполняемые файлы, в которых этот модуль используется. Данный недостаток усугубляется тем фактом, что системному администратору необходимо знать, с какими приложениями скомпонована библиотека

Для устранения представленных недочетов были придуманы **разделяемые** (**динамические**) **библиотеки**

Их ключевая идея состоит в том, что **одна копия объектного модуля разделяется между всеми программами, задействующими его**. **Объектные модули не копируются в компонуемый исполняемый файл; вместо этого единая копия библиотеки загружается в память при запуске первой программы, которой требуются ее объектные модули. Если позже будут запущены другие программы, использующие эту разделяемую библиотеку, они обращаются к копии, уже загруженной в память.**

В операционной системе Windows подобные библиотеки называются **Dynamic Link Library (DLL)**

В операционных системах семейства Linux подобные библиотеки называются **разделяемые объекты (shared objects)**

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)**)**

Некоторые из причин использования DLL в ОС Windows:

* Расширение функциональности приложения
* Возможность использования разных языков программирования
* Более простое управление проектом
* Экономия памяти
* Разделение ресурсов
* Упрощение локализации
* Решение проблем, связанных с особенностями различных платформ
* Реализация специфических возможностей

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 дополнительные 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!**

Если для используемой библиотеки используется только её имя (нет полного пути), то **поиск DLL осуществляется в следующей последовательности**:

* Каталог, содержащий EXE-файл
* Текущий каталог процесса
* Системный каталог Windows
* Основной каталог Windows
* Каталоги, указанные в переменной окружения PATH

[**\_\_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)

* Если он указан перед переменной, прототипом функции или C++-классом, компилятор 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. Чтобы компоновщик клиентского кода мог разрешить символы.**
* 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. Таблица экспорта нужна **загрузчику ОС** на этапе **запуска/загрузки DLL**, чтобы найти **точный адрес** искомого внутри DLL.

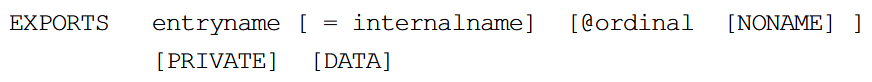
**Посмотреть таблицу экспорта:** **dumpbin -exports path**

**Посмотреть таблицу импорта: dumpbin -imports path**

У каждой функции в таблице экспорта 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**)

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

Чтобы экспортировать по порядковому номеру придётся применить альтернативный способ – **.DEF-файл (файл определений)**



* **entryname**: **Обязательно.** Имя символа, как оно будет известно *снаружи* DLL. Это то имя, которое будет использоваться при связывании по имени. **Важно:** Для функций C++ это должно быть декорированное (mangled) имя, если не используется extern "C".
* = internal\_name: (Опционально) Позволяет задать внутреннее имя символа в коде, если оно отличается от entryname. Редко нужно.
* @ordinal: (Опционально) Явно назначает порядковый номер этому экспорту. ordinal — это целое положительное число. Если не указано, компоновщик назначит номер автоматически.
* NONAME: (Опционально) Экспортировать символ только по порядковому номеру (@ordinal должен быть указан). Имя entryname не будет добавлено в таблицу имен экспорта DLL. Связывание по имени будет невозможно, только по номеру.
* PRIVATE: (Опционально) Ключевое слово PRIVATE запрещает размещение экспортируемого имени в библиотеке импорта..
* DATA: (Опционально) Указывает, что экспортируется переменная (данные), а не код (функция).

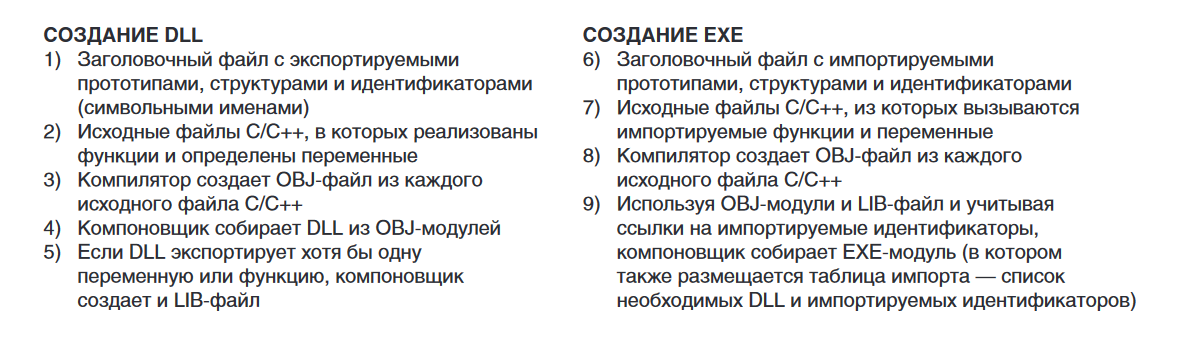
Если мы говорим о создании библиотек на языках отличных от С, то скорее всего при создании символов для таблицы экспорта/импорта компилятор преобразует имена из кода в понятные для себя имена. Данный процесс называется **name decoration или name mangling** (компилятор C такого не делает, он сохраняет имя функций). **.def-файл** позволяет оставить имена функций в том виде, который понятен человеку. Альтернативным способом сохранения имени функции будет модификатор **extern “C”.**

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.

* **extern:** Это стандартное ключевое слово C/C++. Оно сообщает компилятору: "Этот идентификатор (переменная или функция) определен где-то еще. Не выделяй под него память здесь, просто знай, что он существует. Линковщик (компоновщик) найдет его определение позже". Это "где-то еще" может быть:

Разрешая ссылки на импортируемые идентификаторы, компоновщик создает в конечном EXE-модуле раздел или таблицу импорта (imports section). В нем перечисляются DLL, необходимые этому модулю, и идентификаторы, на которые есть ссылки из всех используемых DLL.

Неявный способ:



**I. Подготовка со стороны DLL:**

1. **Заголовочный файл (.h)**: Создается файл, описывающий "публичный интерфейс" DLL. В нем объявляются функции, структуры и переменные, которые DLL будет предоставлять другим модулям (экспортировать). Эти объявления нужны, чтобы другие программы знали, *как* использовать ресурсы DLL. Обычно используется \_\_declspec(dllexport).
2. **Исходные файлы (.cpp)**: Здесь пишется сам код функций и определяются переменные, объявленные в заголовочном файле. Это "внутренности" DLL.
3. **Компиляция (.obj)**: Каждый .cpp-файл компилируется в объектный файл (.obj). Это промежуточное представление кода.
4. **Компоновка DLL**: Компоновщик (linker) собирает все .obj-файлы, относящиеся к DLL, в единый исполняемый файл библиотеки – саму DLL.
5. **Создание LIB-файла**: **Это критически важный шаг для неявного связывания!** Если DLL что-то экспортирует, компоновщик создает *импортную библиотеку* (.lib файл). Этот файл **не содержит кода** DLL, но содержит информацию, необходимую компоновщику *других* программ (например, вашего EXE):
   * Список экспортируемых из DLL функций и переменных.
   * Указание, в каком именно DLL-файле находятся эти экспортируемые элементы.

**II. Сборка EXE с использованием DLL:**

1. **Заголовочный файл (.h)**: Программист EXE включает в свой проект тот же самый (или совместимый) заголовочный файл, который использовался при создании DLL. Но теперь объявления в нем обычно помечаются как импортируемые (например, с \_\_declspec(dllimport)).
2. **Исходные файлы (.cpp)**: В коде EXE программист просто вызывает функции или использует переменные, объявленные в заголовочном файле (Шаг 6), как будто они являются частью самого EXE.
3. **Компиляция (.obj)**: Компилятор обрабатывает .cpp-файлы EXE. Встречая вызов импортируемой функции, он понимает, что ее реализация находится где-то снаружи, и оставляет "заглушку" или пометку для компоновщика.
4. **Компоновка EXE**: **Ключевой этап неявного связывания!**
   * Компоновщик берет .obj-файлы.
   * Ему **обязательно** предоставляется **LIB-файл**.
   * Компоновщик находит в .obj-файлах EXE ссылки на внешние (импортируемые) функции/переменные.
   * Он ищет определения этих ссылок в предоставленном **LIB-файле**.
   * LIB-файл сообщает компоновщику: "Функция DoSomething находится в файле MyLibrary.dll".
   * На основе этой информации компоновщик **встраивает в конечный EXE-файл специальную секцию – Таблицу импорта**. В этой таблице записывается:
     + Список имен всех DLL, от которых зависит этот EXE (например, MyLibrary.dll).
     + Для каждой DLL – список имен функций/переменных, которые EXE из нее импортирует (например, DoSomething).
   * Компоновщик "разрешает" ссылки в коде EXE, направляя вызовы импортируемых функций через записи в Таблице Адресов Импорта (IAT), которая является частью Таблицы Импорта.

Linux при запуске программы ищет .so библиотеки **только** в стандартных системных каталогах (/lib, /usr/lib и т.д.) или путях, указанных специальным образом (например, LD\_LIBRARY\_PATH, rpath). Он **не ищет** их в текущем каталоге или каталоге программы по умолчанию, поэтому программа не находит библиотеку, даже если она лежит рядом.

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

В Linux таблица экспорта и импорта: **readelf -s file**

1. readelf -s выводит содержимое таблицы (или таблиц) символов ELF-файла. Каждая строка в этой таблице описывает один символ (функцию, переменную и т.д.).
2. **Экспортируемые символы:** Это символы, которые **определены** внутри данного файла (.so или исполняемого) и предназначены для использования другими модулями. В выводе readelf -s они обычно имеют:
   1. Поле Ndx (индекс секции): Не равно UND (Undefined). Оно указывает на номер секции внутри файла, где символ определен (например, секция .text для кода, .data для данных).
   2. Поле Bind: Обычно GLOBAL (или WEAK). GLOBAL означает, что символ виден снаружи.
3. **Импортируемые символы:** Это символы, которые **не определены** в данном файле, но используются им. Определение ожидается в другой библиотеке во время динамической линковки. В выводе readelf -s они обычно имеют:
   1. Поле Ndx (индекс секции): Равно UND (Undefined).
   2. Поле Bind: Обычно GLOBAL.

Явная загрузка DLL:

* Подключать заголовочный файл не требуется
* Появились вызовы функций API для работы с DLL
* При сборке приложения больше не требуется указывать использование DLL
* Если вызвать утилиту dumpbin, то для текущего примера мы не найдём в таблице импорта нашей библиотеки, так как ей не откуда там взяться, ведь компоновщик ничего не знает о библиотеке в данном случае

**1. Создание DLL:**

* **Заголовочный файл:** Нужен для самой DLL, чтобы её исходные файлы знали о прототипах экспортируемых функций и т.д. (часто содержит \_\_declspec(dllexport) или используется вместе с .def файлом).
* **Исходные файлы:** Содержат реализацию функций/переменных DLL.
* **Компиляция:** Стандартно, каждый .cpp превращается в .obj.
* **Компоновка DLL:** Стандартно, компоновщик собирает DLL из .obj.
* **Создание LIB-файла:** **Важный момент:** компоновщик MSVC *всё равно создаст* файл библиотеки импорта (.lib), если обнаружит экспортируемые символы (через \_\_declspec(dllexport) или .def файл). **Однако**, как правильно указано в тексте, этот .lib файл **не будет использоваться** тем EXE-файлом, который планирует связываться с этой DLL *явно*. LIB-файл нужен только для *неявного* связывания.

**2. Создание EXE (для Явного связывания):**

* **Заголовочный файл:** Обычно при явном связывании вы **не включаете** заголовочный файл DLL напрямую, чтобы получить прототипы с \_\_declspec(dllimport). Вместо этого **вы определяете типы указателей на функции**, которые соответствуют сигнатурам функций в DLL, которые вы собираетесь загружать. Эти typedef могут быть в вашем заголовочном файле EXE.
* **Исходные файлы:** **Ключевой момент:** В коде EXE **нет прямых вызовов** функций из DLL по их именам (например, нет строк вида result = MyDllFunction(x);). Вместо этого код будет содержать **вызовы LoadLibrary("MyDll.dll") для загрузки DLL и GetProcAddress(hDll, "MyDllFunction") для получения адреса функции во время выполнения**. Полученный адрес сохраняется в указатель на функцию, и вызов происходит через этот указатель.
* **Компиляция:** Стандартно, каждый .cpp превращается в .obj.
* **Компоновка EXE:**
  + **LIB-файл DLL не нужен:** Поскольку в коде EXE нет прямых ссылок на символы из DLL, компоновщик не видит "неразрешенных внешних символов" (unresolved external symbols), относящихся к этой DLL. Ему не нужна библиотека импорта (.lib) для их разрешения.
  + **Раздел импорта:** отсутствует.

