1 задача

`Написать программу hello.c, которая выводит фразу “Hello world”:` a. получить исполняемый файл;
`b. посмотреть unresolved symbols (puts, printf) с помощью nm; c. посмотреть зависимости (ldd);

Команда. `nm` используется для просмотра символов в объектных файлах и исполняемых программах.

Символы в выводе представляют собой *функции*, *переменные* и *другие объекты*, которые используются в программе. Каждая строка содержит информацию о символе, его типе и адресе (или значении).

```
<адрес> <тип> <имя_символа>
```

Типы символов:

- **T** (Text) — символ находится в секции кода (например, функции).
- **t** (static Text) — статическая функция (локальная для модуля).
- **D** (Data) — символ находится в секции данных (глобальные переменные).
- **d** (static Data) — статическая переменная (локальная для модуля).
- **B** (BSS) — неинициализированные глобальные переменные.
- **b** (static BSS) — неинициализированные статические переменные.
- **R** (Read-only) — символ находится в секции только для чтения (например, строковые константы).
- **r** (static Read-only) — статические данные только для чтения.
- **U (Undefined) — символ не определен в текущем объектном файле (например, функции из внешних библиотек).**
- **w** (Weak) — слабый символ (может быть переопределен).
- **V** (Weak object) — слабый объект.
- **A** (Absolute) — абсолютный символ (не связан с секцией).
- **C** (Common) — неинициализированные глобальные переменные (устаревший тип).
- **?** — неизвестный тип символа.

Unresolved symbols — это символы (функции, переменные), которые компилятор не смог найти во время сборки программы. Это может произойти, если вы используете функцию из другой библиотеки, но не подключили эту библиотеку при сборке.

Символы **не разрешимы**, это означает, что компилятор или линковщик не смогли найти определение этих символов в текущей единице трансляции (т.е. в вашем исходном коде или в уже собранных объектных файлах). Эти символы могут находиться в других библиотеках, которые еще не были связаны с вашим проектом.

```
000000000000038c r __abi_tag
0000000000004010 B __bss_start
0000000000004010 b completed.0
                 w __cxa_finalize@GLIBC_2.2.5
0000000000004000 D __data_start
0000000000004000 W data_start
0000000000001090 t deregister_tm_clones
0000000000001100 t __do_global_dtors_aux
0000000000003dc0 d __do_global_dtors_aux_fini_array_entry
0000000000004008 D __dso_handle
0000000000003dc8 d _DYNAMIC
0000000000004010 D _edata
0000000000004018 B _end
000000000000116c T _fini
0000000000001140 t frame_dummy
0000000000003db8 d __frame_dummy_init_array_entry
00000000000020f0 r __FRAME_END__
0000000000003fb8 d _GLOBAL_OFFSET_TABLE_
                 w __gmon_start__
0000000000002010 r __GNU_EH_FRAME_HDR
0000000000001000 T _init
0000000000002000 R _IO_stdin_used
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable
                 U __libc_start_main@GLIBC_2.34
0000000000001149 T main
                 U printf@GLIBC_2.2.5
00000000000010c0 t register_tm_clones
0000000000001060 T _start
0000000000004010 D __TMC_END__
```

Число `0000000000004010(например)` в выводе команды `nm` или других инструментов — это **адрес** символа в памяти или в исполняемом файле. Этот адрес относится к **виртуальной памяти**, которая выделяется программе при ее запуске. В контексте исполняемого файла (например, `a.out` или `hello`), это адрес внутри одной из секций файла. Секции файла это его сегменты

- **`.text`** — секция кода (машинные инструкции).
- **`.data`** — секция инициализированных данных.
- **`.bss`** — секция неинициализированных данных (Block Started by Symbol).
- **`.rodata`** — секция данных только для чтения (например, строковые константы).

- **T** (Text) — символ находится в секции кода (например, функции).

- **U (Undefined)** — символ не определен в текущем объектном файле (например, функции из внешних библиотек).

- **puts**: Функция из стандартной библиотеки C, которая выводит строку текста на экран и автоматически добавляет символ новой строки (`\n`) после вывода.
- **printf**: Более гибкая функция для форматированного вывода данных на экран. Позволяет выводить текст, числа, символы и другие данные с использованием специальных форматов.

Когда происходит препроцессинг в место метки `#include <stdio.h>` в код подставляются объявленные функции в этой библиотеке.

Заголовочные файлы не содержат реализацию функций. Они нужны для того, чтобы компилятор знал, как вызывать функции (например, какие типы аргументов они принимают и какой тип возвращают).

```
int printf(const char *format, ...);
```

Линкер же подставляет реализацию функций(находятся в библиотеках) в результирующий код.

a) **Динамическая линковка:**

- При динамической линковке код библиотек (например, `libc.so`) **не включается** в исполняемый файл. Вместо этого исполняемый файл **содержит ссылки на эти библиотеки.**
- Во время выполнения программы операционная система загружает библиотеки (например, `libc.so`) в память и связывает их с вашей программой.
- Преимущества:
  - Исполняемый файл меньше по размеру.
  - Несколько программ могут использовать одну и ту же библиотеку в памяти.
- Недостатки:
  - Программа зависит от наличия библиотек в системе.

b) **Статическая линковка:**

- При статической линковке код библиотек (например, `libc.a`) **включается в исполняемый файл.**
- Преимущества:
  - Программа не зависит от библиотек в системе.
- Недостатки:
  - Исполняемый файл больше по размеру.

```
#include <stdio.h>

int main() {
    fprintf(stdout, "Hello, World!\n");
    return 0;
}
```

--------- Как выбирается тип библиотеки?-------

По умолчанию компилятор (`gcc`) использует **динамическую линковку** (`libc.so`). Однако вы можете явно указать, какую библиотеку использовать.

a) Динамическая линковка (по умолчанию):

```
gcc main.c -o program
```

- В этом случае линкер использует `libc.so`.

b) Статическая линковка:

```
gcc -static main.c -o program
```

- В этом случае линкер использует `libc.a`, и код библиотеки включается в исполняемый файл.

С помощью команды `file programm` можно проверить какая библиотека в данный момент подключена к вашей программе.

- **При статической линковке**: Вся библиотека (или только те части, которые используются) будет включена в исполняемый файл на этапе линковки. То

есть после завершения работы линковщика ваш исполняемый файл будет содержать всю необходимую реализацию.
- **При динамической линковке**: Библиотека никогда не будет включена в исполняемый файл. Вместо этого исполняемый файл будет содержать только ссылки на функции, а сама библиотека будет загружена в память во время выполнения программы.

`ldd programm`:
Ваша программа `a.out` зависит от нескольких стандартных системных библиотек Linux:

1. **linux-vdso.so.1**: Это виртуальная динамическая общая библиотека (Virtual Dynamic Shared Object), предоставляемая ядром операционной системы. Она содержит высокоэффективные системные вызовы, которые работают быстрее благодаря тому, что выполняются непосредственно в пользовательском пространстве, минуя переход в режим ядра.
   (Вместо того чтобы выполнять дорогостоящий переход в режим ядра для системных вызовов (например, `gettimeofday`), `vdso` позволяет выполнять некоторые системные вызовы в пользовательском пространстве, что ускоряет их выполнение.)
2. **libc.so.6**: Это стандартная библиотека языка C (GNU C Library), которая предоставляет основные функции, необходимые для работы программ на языке C, такие как ввод-вывод, работа с памятью, математические операции и многое другое. Версия `6` указывает на конкретную версию этой библиотеки.
   (Эта библиотека предоставляет базовые функции для работы с памятью, строками, вводом-выводом, математическими операциями и другими задачами. /lib/x86_64-linux-gnu/libc.so.6 - путь)
3. **ld-linux-x86-64.so.2**: Это динамический линкер, который отвечает за загрузку и выполнение динамически связанных библиотек. Он загружается вместе с программой и обеспечивает корректное разрешение всех необходимых символов из внешних библиотек.

Статические и динамические библиотеки — это два способа организации и использования кода, который может быть общим для множества программ. Они различаются по способу включения в программу и управлению ими во время исполнения. Рассмотрим оба типа библиотек, их отличия, преимущества и недостатки.

----------Что это говорит о вашей программе?---------

- Ваша программа использует **динамическую линковку**. Это означает, что она зависит от внешних библиотек (`libc.so.6` и других), которые должны быть установлены в системе.
- Программа использует стандартную библиотеку C (`libc.so.6`) для выполнения базовых операций (например, ввод-вывод, работа с памятью).
- Динамический загрузчик (`ld-linux-x86-64.so.2`) будет загружать библиотеки и связывать их с вашей программой при ее запуске.

Когда вы запускаете программу (`a.out`), происходит следующее:

1. Операционная система загружает исполняемый файл и динамический загрузчик (`ld-linux-x86-64.so.2`).
2. Динамический загрузчик находит и загружает все необходимые библиотеки (например, `libc.so.6`).
3. Загрузчик связывает вызовы функций в вашей программе (например, `printf`) с их реализациями в библиотеках.
4. Программа начинает выполнение.

Статические библиотеки

Определение: Статическая библиотека — это архивный файл, содержащий объектные файлы, которые компонуются (линкуются) с основной программой на этапе компиляции. В результате получается единый исполняемый файл, содержащий весь необходимый код.

Форматы: На Unix-подобных системах статические библиотеки обычно имеют расширение `.a` (от "archive"), а на Windows — `.lib` .

Плюсы статических библиотек:

- **Самодостаточность:** Исполняемый файл содержит всё необходимое для своей работы, что упрощает распространение программы. Вам не нужно беспокоиться о наличии нужных библиотек на целевой системе.
- **Производительность:** Поскольку вся необходимая логика содержится прямо в исполняемом файле, обращение к функциям из библиотеки происходит

быстрее, так как отсутствуют накладные расходы на динамическое связывание.
- **Контроль версий:** Все зависимости жестко зафиксированы на момент компиляции, что уменьшает вероятность проблем с несовместимостью версий библиотек.

Минусы статических библиотек:

- **Размер исполняемого файла:** Размер исполняемого файла увеличивается, так как в него включается весь код из библиотеки, даже если используются лишь отдельные функции.
- **Обновление:** Если требуется обновить библиотеку, необходимо перекомпилировать всю программу заново. Это может быть неудобно для больших проектов.
- **Дублирование кода:** Если несколько программ используют одну и ту же статическую библиотеку, каждая из них будет содержать копию одного и того же кода, что увеличивает общее потребление ресурсов.

Динамические библиотеки

Определение: Динамическая библиотека — это отдельный файл, который загружается и связывается с программой во время её выполнения. Код из динамической библиотеки не встраивается напрямую в исполняемый файл, а используется по мере необходимости.

Форматы: На Unix-подобных системах динамические библиотеки обычно имеют расширение `.so` ("shared object"), а на Windows — `.dll` ("dynamic-link library").

Плюсы динамических библиотек:

- **Экономия места:** Несколько программ могут использовать одну и ту же динамическую библиотеку, что снижает общий объем занимаемого дискового пространства.
- **Упрощённое обновление:** Можно обновлять библиотеку отдельно от программы, что облегчает поддержку и исправление ошибок.
- **Модульность:** Программы могут загружать только нужные им части библиотеки, что улучшает производительность и уменьшает использование памяти.

Минусы динамических библиотек:

- **Зависимость:** Программе требуются внешние файлы (динамические библиотеки), которые должны быть установлены на целевой системе. Это усложняет распространение программы.
- **Проблемы совместимости:** Могут возникнуть проблемы с версиями библиотек, особенно если на целевой системе установлена устаревшая или несовместимая версия.
- **Производительность:** Из-за необходимости динамического связывания на этапе выполнения, программы могут работать немного медленнее по сравнению с теми, которые используют статические библиотеки.

Отличие между статическими и динамическими библиотеками

- **Время связывания:**
  - **Статические библиотеки:** Связываются на этапе компиляции.
  - **Динамические библиотеки:** Связываются во время выполнения программы.
- **Размер исполняемого файла:**
  - **Статические библиотеки:** Увеличивают размер исполняемого файла.
  - **Динамические библиотеки:** Оставляют исполняемый файл компактным, так как код библиотеки находится в отдельном файле.
- **Обновляемость:**
  - **Статические библиотеки:** Требуют полной перекомпиляции программы для обновления.
  - **Динамические библиотеки:** Могут быть обновлены независимо от программы.
- **Совместное использование:**
  - **Статические библиотеки:** Каждая программа содержит свою собственную копию кода библиотеки.
  - **Динамические библиотеки:** Одна копия библиотеки может использоваться несколькими программами одновременно.

**Написать статическую библиотеку с функцией hello_from_static_lib() и использовать ее в hello.c:

a. посмотреть исполняемый файл на предмет того будет ли функция hello_from_static_lib() unresolved. Почему?

b. где находится код этой функции?**

-----Как написать статическую библиотеку?---

https://firststeps.ru/linux/r.php?5
(алгоритм для чайников)

1. Создадим файл `hello_lib.c` с функцией `hello_from_static_lib`:

```
#include <stdio.h>

void hello_from_static_lib() {
    printf("Hello from static library!\n");
}
```

2. Скомпилируем `hello_lib.c` в объектный файл `hello_lib.o`:

```
gcc -c hello_lib.c -o hello_lib.o
```

3. Создадим статическую библиотеку `libhello.a` из объектного файла:

```
ar rcs libhello.a hello_lib.o
```

где

- `ar` — утилита для создания архивов (статических библиотек).
- `rcs` — флаги:
  - `r` — добавить файл в архив (или заменить, если уже существует).
  - `c` — создать архив, если он не существует.

```
- `s` — добавить индекс символов в архив.
```

Мы умнички создали статическую библиотеку!!

Использование статической библиотеки в `hello.c`

1. Создадим файл `hello.c`, который будет использовать функцию `hello_from_static_lib`:

```
void hello_from_static_lib(); // Объявление функции

int main() {
    hello_from_static_lib(); // Вызов функции из статической библиотеки
    return 0;
}
```

2. компилируем `hello.c` и свяжем его с нашей статической библиотекой `libhello.a`:

```
gcc hello.c -L. -lhello -o hello
```

где:

- `-L.` — указывает компилятору искать библиотеки в текущей директории (`.`).
- `-lhello` — указывает компилятору использовать библиотеку `libhello.a`.

3. Используем команду `nm`, чтобы посмотреть символы в исполняемом файле `hello`:

```
nm hello
```

```
0000000000001162 T hello_from_static_lib
```

Код функции `hello_from_static_lib` находится **внутри исполняемого файла `hello`**. Это произошло потому, что мы использовали статическую библиотеку (`libhello.a`), и компилятор включил код функции в исполняемый файл.

-----Как написать динамическую библиотеку?---

https://firststeps.ru/linux/r.php?6
(так же алгоритм для чайников)

1. Создадим файл `hello_dynamic_lib.c`:

```
#include <stdio.h>
```

```
void hello_from_dynamic_lib() {
    printf("Hello from dynamic library!\n");
}
```

2. Скомпилируем `hello_dynamic_lib.c` в динамическую библиотеку `libhello.so`:

```
gcc -shared -fPIC hello_dynamic_lib.c -o libhello.so
```

где:

- `-shared` — указывает компилятору создать динамическую библиотеку.
- `-fPIC` — генерировать позиционно-независимый код (Position Independent Code), который необходим для динамических библиотек.

Теперь у нас есть динамическая библиотека `libhello.so`.
Мы так же умнички!

3. Использование динамической библиотеки в `hello.c`

```
void hello_from_dynamic_lib(); // Объявление функции

int main() {
    hello_from_dynamic_lib(); // Вызов функции из динамической библиотеки
    return 0;
}
```

4. Скомпилируем `hello.c` и свяжем его с динамической библиотекой `libhello.so`:

```
gcc hello.c -L. -lhello -o hello
```

5. Проверим состояние функции `hello_from_dynamic_lib`
`nm hello.c`

```
 U hello_from_dynamic_lib
```

`ldd hello.c`

```
libhello.so => not found
```

Мы видим, что `libhello.so` не найдена (`not found`). Это связано с тем, что система не знает, где искать нашу библиотеку.
1. Чтобы система могла найти `libhello.so` , нужно указать путь к библиотеке. Это можно сделать двумя способами:
1. 1. **Добавить путь в `LD_LIBRARY_PATH`** :
-

```
export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
   ./hello
```

- ```
  1. 2 **Установить библиотеку в системный путь:
  ```

  Скопируйте libhello.so в системный каталог, например /usr/lib или /usr/local/lib:

```
sudo cp libhello.so /usr/local/lib
sudo ldconfig
./hello
```

так же делаем проверку! через ldd nm.

Динамическая библиотека с `hello_from_dyn_runtime_lib()` и `dlopen`

`dlopen` — это функция из стандартной библиотеки `libdl` , которая позволяет **динамически загружать разделяемые библиотеки** (shared libraries, например, `.so` на Linux или `.dll` на Windows) во время выполнения программы. Это означает, что библиотека не обязательно должна быть связана с программой на этапе компиляции, а может быть загружена в любой момент во время работы программы.
ГРУБО ГОВОРЯ Представьте, что у вас есть программа, и вы хотите добавить в неё новые функции. Обычно вы бы добавили эти функции прямо в код программы, перекомпилировали её, и всё готово. Но что, если вы хотите, чтобы программа могла добавлять новые функции **без перекомпиляции**? Например, чтобы вы могли добавлять плагины или модули, которые программа будет загружать только тогда, когда они нужны.
```

Вот тут и появляется `dlopen`. Это функция, которая позволяет вашей программе **загружать дополнительные куски кода** (библиотеки) прямо во время работы программы. Эти куски кода хранятся в файлах с расширением `.so`

Почему это удобно?

1. **Гибкость**:
   - Вы можете добавлять новые функции в программу без её перекомпиляции.
   - Например, программа может загружать разные плагины в зависимости от того, что нужно пользователю.
2. **Экономия памяти**:
   - Если библиотека не нужна, её можно не загружать, что экономит память.
3. **Обновления**:
   - Вы можете обновлять библиотеки отдельно от основной программы.
     Синтаксис:

Синтаксис:

```
void *dlopen(const char *filename, int flags);
```

- `filename` — путь к библиотеке (например, `"./libhello.so"`).
- `flags` — флаги, определяющие, как библиотека будет загружена (например, `RTLD_LAZY` или `RTLD_NOW`).

1. Создание динамической библиотеки

a) Создадим файл `hello_runtime_lib.c`:

```
#include <stdio.h>

void hello_from_dyn_runtime_lib() {
    printf("Hello from runtime-loaded dynamic library!\n");
}
```

b) Скомпилируем `hello_runtime_lib.c` в динамическую библиотеку `libruntimehello.so`:

```
gcc -shared -fPIC hello_runtime_lib.c -o libruntimehello.so
```

---

2. **Использование `dlopen` в `hello.c`**

a) Создадим файл `hello.c`, который будет загружать библиотеку `libruntimehello.so` во время выполнения с помощью `dlopen`:

```
// hello.c
#include <stdio.h>
#include <dlfcn.h> // Для dlopen, dlsym, dlclose

int main() {
    void *handle;
    void (*hello_from_dyn_runtime_lib)();

    // Открываем динамическую библиотеку
    handle = dlopen("./libruntimehello.so", RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "Error: %s\n", dlerror());
        return 1;
    }

    // Получаем адрес функции
    hello_from_dyn_runtime_lib = dlsym(handle,
"hello_from_dyn_runtime_lib");
    if (!hello_from_dyn_runtime_lib) {
        fprintf(stderr, "Error: %s\n", dlerror());
        dlclose(handle);
        return 1;
    }

    // Вызываем функцию
    hello_from_dyn_runtime_lib();

    // Закрываем библиотеку
    dlclose(handle);
    return 0;
```

```
}
```

Разбираем программу по строчкам!

* `*void* handle; void (*hello_func)();`

`void *handle;`

- Это указатель на "дескриптор" загруженной библиотеки. После вызова `dlopen`, функция возвращает указатель на загруженную библиотеку, который используется для дальнейшей работы с ней (например, для поиска символов с помощью `dlsym` или выгрузки с помощью `dlclose`).
- Тип `void *` означает, что это указатель на что-то неизвестного типа (в данном случае — на загруженную библиотеку).

`void (*hello_func)();`

- Это объявление указателя на функцию.
- `void (*hello_func)();` означает, что `hello_func` — это указатель на функцию, которая не принимает аргументов (`()`) и возвращает `void` (ничего).
- После вызова `dlsym`, этот указатель будет указывать на функцию `hello_from_dyn_runtime_lib` из загруженной библиотеки.

`handle = dlopen("./libhello.so", RTLD_LAZY);`

- Библиотека загружается в **адресное пространство текущего процесса**. Это означает, что код библиотеки становится доступным для выполнения в рамках вашей программы.
- Операционная система выделяет память для загрузки библиотеки и связывает её с вашим процессом.

Почему `RTLD_LAZY`?

- `RTLD_LAZY` — это флаг, который указывает, что символы (например, функции или переменные) из библиотеки будут разрешаться (загружаться) только при первом использовании.
- Это полезно для оптимизации: если какие-то функции из библиотеки никогда не вызываются, то их разрешение не происходит, что экономит время и

ресурсы.

Что будет, если не `LAZY`?

- Если использовать флаг `RTLD_NOW` вместо `RTLD_LAZY`, то **все символы из библиотеки будут разрешены сразу при загрузке**.
- Это может быть полезно, если вы хотите сразу проверить, что все необходимые символы доступны, и получить ошибку, если что-то отсутствует.
- Однако это может замедлить загрузку библиотеки, особенно если она большая и содержит много символов.

```
hello_func = dlsym(handle, "hello_from_dyn_runtime_lib");
```

`dlsym` — это функция, которая позволяет получить указатель на символ (например, функцию или переменную) из загруженной библиотеки.

Синтаксис:

```
void *dlsym(void *handle, const char *symbol);
```

- `handle` — дескриптор библиотеки, возвращённый `dlopen`.
- `symbol` — имя символа (например, имя функции), который вы хотите найти.

Что делает `dlsym`?

- Ищет символ (например, функцию `hello_from_dyn_runtime_lib`) в загруженной библиотеке.
- Возвращает указатель на этот символ. Если символ не найден, возвращает `NULL`.

b) Скомпилируем `hello.c` с поддержкой `dlopen`:

```
gcc hello.c -ldl -o hello
```

- `-ldl` — подключает библиотеку `libdl`, которая предоставляет функции `dlopen`, `dlsym`, `dlclose`.

---

3. **Анализ исполняемого файла**

a) Используем команду `nm`, чтобы посмотреть символы в исполняемом файле `hello` Используем команду `ldd`:

```
nm hello
```

- Мы не увидим символ `hello_from_dyn_runtime_lib`, потому что он загружается динамически во время выполнения.
  В вашем коде библиотека `libhello.so` загружается **динамически во время выполнения программы** с помощью `dlopen`. Это означает, что символы из этой библиотеки (например, функция `hello_from_dyn_runtime_lib`) **не включаются в исполняемый файл** `hello` на этапе компиляции и линковки.
- Вместо этого, программа ищет и загружает эти символы **во время выполнения**.

b) **Проверим динамические зависимости**

Вывод будет примерно таким:

```
linux-vdso.so.1 (0x00007ffe8ffed000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007b0907a00000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007b0907a00000)
/lib64/ld-linux-x86-64.so.2 (0x00007b0907cf6000)
```

- Мы видим только `libdl.so.2` и `libc.so.6`, потому что `libruntimehello.so` загружается динамически.

---

4. **Запуск программы**

Теперь программа должна работать корректно:

```
./hello
```

Вывод:

```
Hello from runtime-loaded dynamic library!
```

5. Что происходит при использовании `dlopen`?

- `dlopen` загружает динамическую библиотеку во время выполнения программы
- `dlsym` получает адрес функции из библиотеки.
- `dlclose` закрывает библиотеку после использования.

Динамические библиотеки (например, `.so` в Linux или `.dll` в Windows) — это скомпилированные модули кода, которые могут быть загружены и использованы программой во время выполнения. Они предоставляют функции, которые могут быть вызваны основной программой.
API (Application Programming Interface) — **это набор правил и инструкций, с помощью которых различные приложения и сервисы могут взаимодействовать друг с другом**. [1]

По сути API — это **посредник**, который позволяет одной программе «общаться» с другой, обмениваться нужной датой и отображать её для пользователей.

**Плагины — это особый тип динамических библиотек, которые предназначены для расширения функциональности конкретной программы. Они обычно следуют определенному интерфейсу или API, предоставляемому основной программой.
Основные характеристики:

- **Расширение функциональности**: Плагины добавляют новые возможности в программу, например, поддержку новых форматов файлов, дополнительных функций и т.д.
- **Специфичность**: Плагины обычно разрабатываются для конкретной программы или платформы и следуют ее API.
- **Гибкость**: Плагины позволяют пользователям или разработчикам добавлять новые функции без необходимости перекомпиляции основной программы.

Вопросы: gcc - использует puts ? анализированные символы(символы которые нужны при загрузки программы. что за puts(как он решил, этапы компиляции). чтобы был puts нужно \n. puts откуда он взялся(??) puts- ставит препроцессинг. (неразрешенные символы) Вывод зависимости: Зависимости что это? Что делает Инклюд? На каком этапе работают динамические библиотеки? Что делает препроцессор?
Что находится в библиотеках?? Научится читать по английски....

Отличия символов при nm . Исходник - код. ссылка при запуске на динамическую библиотеку . зачем так сделано? зачем есть статические библиотеки зачем динамические. Кроме экономии времени , что-то еще.

что такое символы ? что-то про оптимизацию с printf
Зачем dlopen спрашивает по коду. Для чего, почему , зачем.