СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

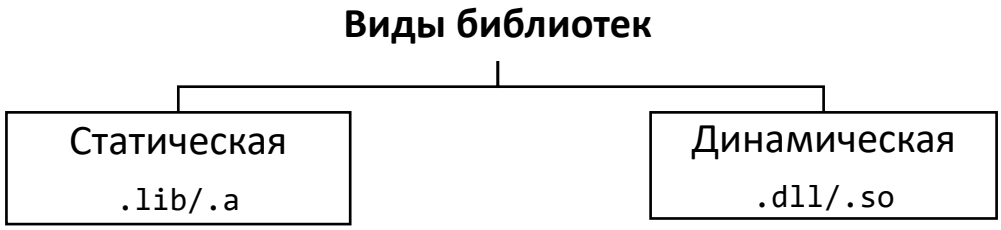
Опорный конспект по теме №2

«Создание и использование библиотек»

1. Виды библиотек

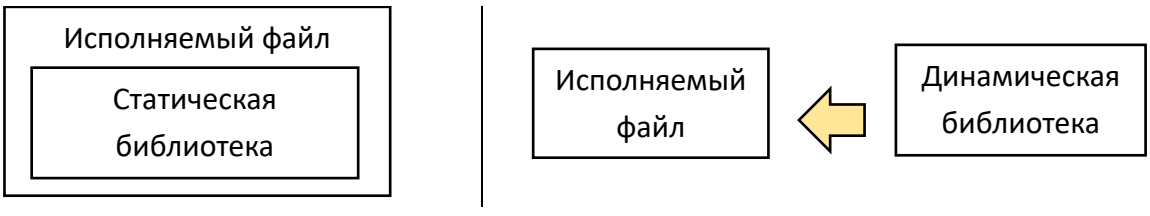
Библиотеками называют файлы объектного кода, скомпилированные с использованием специальных флагов компилятора данного языка программирования и содержащие набор определенных символьных имен (типов и функций), представляемых этой библиотекой для использования в другом файле объектного кода.

Традиционно существует два вида библиотек:



Данные виды библиотек работают совершенно по-разному:

Статическая библиотека	Динамическая библиотека
• Библиотека полностью включается в содержимое результирующего файла как его часть • Символы из библиотеки подгружаются в исполняемый файл на этапе компиляции	• Библиотека не включается в содержимое результирующего файла, но является его зависимостью. • Символы из библиотеки подгружаются в оперативную память на этапе загрузки приложения



Если со статическими библиотеками все понятно, то как работают динамические библиотеки?

Каждый исполняемый файл, поддерживающий работу с динамическими библиотеками, в своем формате имеет таблицу зависимых библиотек. Данные библиотеки можно просмотреть для любого исполняемого файла с помощью команды

```
ldd {файл}
```

За подгрузку этих библиотек при запуске отвечает специальный компонент системы, называемый *программным загрузчиком*. В Linux он находится по пути `/lib64/ld-linux-x86_64.so.2`. Он для всех таких программ также является зависимостью.

Если программный загрузчик не обнаруживает в системе зависимость по некоторым путям, то он приостанавливает загрузку программы и выписывает соответствующую ошибку.

2. Использование библиотек

Библиотеки используются в коде посредством включения их символов, объявленных как типы или функции в заголовочных файлах с расширением .h. Когда пользователь в коде прописывает препроцессорную директиву #include с указанием некоторого заголовочного файла, потенциально этот файл может содержать символы, ссылающиеся на внешнюю библиотеку. Такие символы имеют виды объявлений (прототипов), то есть объектов, не имеющих тела, например:

```
int function(double value);
```

В свою очередь, определение этих символов находится во внешнем файле объектного кода, то есть в самой библиотеке. Для объединения файлов объектных кодов при создании исполняемого файла производится компоновка (линковка) — один их этапов сборки программы.

Внешние библиотеки как правило можно скачать с использованием любого пакетного менеджера (vcpkg/Conan/APT). Например, скачивание библиотеки для работы с БД SQLite с помощью APT:

```
sudo apt install libsqlite3-dev
```

В APT пакеты библиотек всегда имеют префикс `lib` (от англ. *library* — библиотека). Для использования их в разработке (то есть с наличием заголовочных файлов) необходимо искать пакеты, имеющие суффикс `dev` (от англ. *development* — разработка).

Как правило, библиотеки устанавливаются в систему по специальным путям (пути библиотек). Они подразделяются на системные и пользовательские.

- *Системные пути*: `/lib`, `/lib64`, `/usr/lib/`, `/usr/local/lib`.
- *Пользовательские пути*: `~/.local/lib` или текущий каталог проекта.

Для использования библиотек в системе сборке CMake предусмотрена специальная функция `target_link_libraries`, о которой было упомянуто в предыдущем конспекте.

3. Создание статических библиотек

Для создания статических библиотек из исходного кода требуется соблюдать всего несколько условий:

- В файлах исходного кода, предназначенного для создания библиотеки, **не определена** функция с именем `main`, иначе использование этой библиотеки может повлечь

«двусмысленность» имен компилятора (англ. *ambiguity*), так как это имя зарезервировано как точка входа исполняемого файла.

- При использовании системы сборки CMake в файле CMakeLists.txt необходимо создать из перечисленных файлов исходного кода отдельную цель, представляющую собой статическую библиотеку с заданным именем с использованием функции

```
add_library({имя} STATIC {…})
```

- При компиляции с использованием компилятора g++ необходимо указать флаг `-static`. Также можно использовать архиватор, который объединит все файлы объектного кода в один.

Плюсы и минусы использование статических библиотек:

Плюсы	Минусы
• Простота • Быстрая работа программы • Отсутствие зависимостей	• Большой вес программы • Долгая компиляция • Дублирование кода

4. Создание динамических библиотек

Для создания динамических библиотек из исходного кода требуется соблюдать следующие несколько условий:

- В файлах исходного кода, предназначенного для создания библиотеки, **не определена** функция с именем `main`, иначе использование этой библиотеки может повлечь «двусмысленность» имен компилятора (англ. *ambiguity*), так как это имя зарезервировано как точка входа исполняемого файла.
- При использовании системы сборки CMake в файле CMakeLists.txt необходимо создать из перечисленных файлов исходного кода отдельную цель, представляющую собой динамическую библиотеку с заданным именем с использованием функции

```
add_library({имя} SHARED {…})
```

- При компиляции с использованием компиляторов из GCC необходимо указать флаг `-shared`.
- В некоторых случаях может понадобится использование флага `-fPIC` (аббр. *Position Independent Code*, рус. *Код с независимой адресацией*). Так как разделяемые библиотеки могут быть использованы разными процессами, не всегда они могут загрузиться по одному и тому же адресу в памяти, следовательно программный загрузчик должен определять свободные адреса для загрузки библиотеки в момент запуска процесса.

Плюсы и минусы использование динамических библиотек:

Плюсы	Минусы
• Малый вес программы • Быстрая компиляция • Отсутствие дублирования кода	• Более долгий запуск приложения • Имеются обязательные зависимости

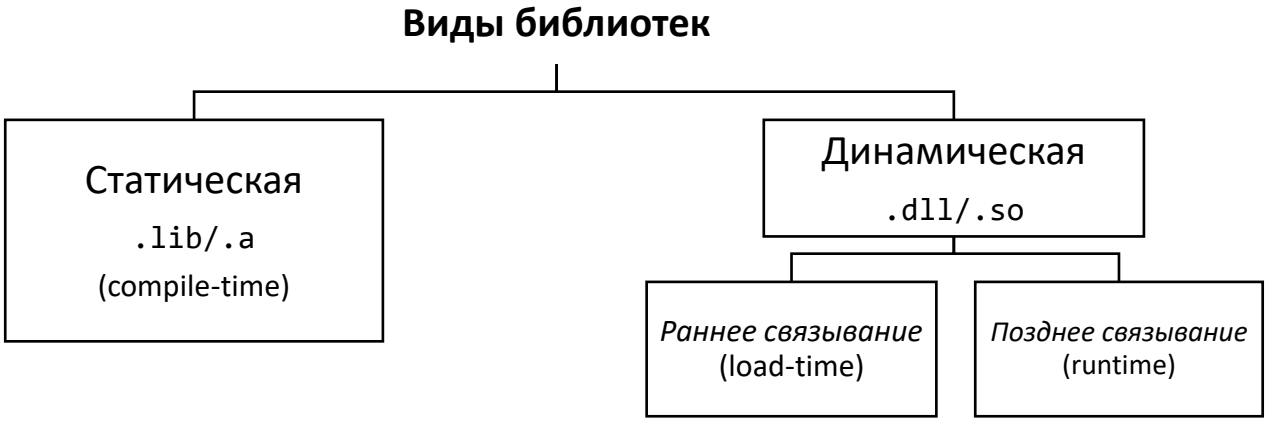
Возникает резонный вопрос: а можно ли использовать динамические библиотеки, но так, чтобы процесс их загрузки можно было контролировать самостоятельно?

Сопутствующие вопросы:

- Можно ли принудительно запустить приложение без зависимостей?
- Можно ли динамические библиотеки подгружать и выгружать в процессе выполнения программы?
- Можно ли обработать отсутствие библиотеки? Например, если она не нашлась, то попытаться работать без нее?

5. Понятие раннего и позднего связывания

Оказывается, использование динамических библиотек подразумевает использование двух видов связывания: *раннее* и *позднее* связывание.



Раннее связывание подразумевает то самое добавление записи в таблицу зависимостей исполняемого файла данной библиотеки, чтобы программный загрузчик на этапе загрузки () мог перенести символы из библиотеки в память процесса.

Позднее связывание, в свою очередь, не формирует зависимостей для исполняемого файла, однако использует библиотеки «неявно», в процессе выполнения программы. Для этого необходимо использовать специальные средства операционных систем, которые осуществляют:

- Загрузку библиотеки по пути/имени в оперативную память.
- Импорт символьных имен из библиотеки в адресное пространство процесса.
- Выгрузку библиотеки из оперативной памяти.

В ядре Linux данные три функцию осуществляет специальная библиотека `libdl.so`, которую в CMake можно скомпоновать с заданной по имени целью, используя функцию

```
target_link_libraries({имя} ${CMAKE_DL_LIBS})
```

Для использования библиотеки в исходном коде необходимо включить заголовочный файл `dlfcn.h`:

```
#include <dlfcn.h>
```

Данный заголовочный файл предоставляет три функции:

```
void* dlopen(const char* name, int flags);
void* dlsym(void* handle, const char* symbol);
void dlclose(void* handle);
```

Первая функция `dlopen` осуществляет загрузку библиотеки в память по имени с использованием заданных флагов (константы RTLD_LAZY или RTLD_NOW — «ленивая» или «жадная» загрузка). Функция возвращает некоторый обобщенный указатель, представляющий собой управляющий объект данной библиотеки для использования в других функциях.

Примечание: функция может распознать как имя библиотеки, так и относительный или абсолютный путь до нее. При использовании имени поиск библиотеки будет осуществлен сначала по путям, обозначенным в переменной окружения `LD_LIBRARY_PATH`, затем в каталогах `/lib` и `/usr/lib`.

Примечание 2: если библиотеку не удалось загрузить, то будет возвращен нулевой указатель (`nullptr`). Сообщение об ошибке можно получить с помощью функции `dlerror`.

Вторая функция `dlsym` осуществляет импорт символа из библиотеки (используя управляющий объект библиотеки) по имени. Возвращаемое значение также представляет собой обобщенный указатель на данный символ, однако на самом деле является указателем на функцию, и поэтому должен быть явно преобразован к типу указателя искомой функции.

Примечание: с этой функцией нужно быть осторожным, так как она может работать не так, как ожидается, и позже этому будет дано объяснение.

Примечание 2: если символ не удалось загрузить, то будет возвращен нулевой указатель (`nullptr`). Сообщение об ошибке можно получить с помощью функции `dlerror`.

Третья функция `dlclose` осуществляет выгрузку библиотеки из памяти, если она больше не нужна.

Пример использования:

```
#include <dlfcn.h>
#include <iostream>

// Тип функции
typedef double (*sqrt_func)(double);

int main()
{
    void* handle = dlopen("libmysqrt.so");
    if (handle == nullptr)
    {
        std::cerr << "Error: " << dlerror() << std::endl;
        return -1;
    }

    auto* function = reinterpret_cast<sqrt_func>(dlsym(handle, "sqrt"));
    if (function == nullptr)
    {
        std::cerr << "Error: " << dlerror() << std::endl;
        dlclose(handle);
```

```
        return -1;
    }

    double value = function(16.0);
    std::cout << "Square root of 16.0 is " << value << std::endl; // 4.0
    dlclose(handle);
    return 0;
}
```

6. Создание собственной динамической библиотеки для позднего связывания

Как было сказано в предыдущем пункте, в функции `dlsym` может возникнуть некоторая сложность.

Рассмотрим пример. Мы создали библиотеку, в которой есть функция вида

```
double func(std::string name);
```

При компиляции разделяемой библиотеки мы получим из этой функции некоторое имя, которое мы ожидаем увидеть как «`func`». Для получения символьных имен из библиотеки воспользуемся командой

```
nm {библиотека}
```

и убедимся, что символ имеет имя

```
__Z4funcNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEE
```

вместо ожидаемого имени «func».

Дело в том, что при компиляции символьных имен из языка C++ они подвергаются специальному процессу под названием *мэнглинг* (от англ. *mangle* — калечить). Он, в первую очередь, предназначен для того, чтобы в символьном имени оставить информацию об аргументах функции и возвращаемом значении, так как C++ поддерживает перегрузку функций, и более того, может иметь одинаковые имена функций в разных классах и пространствах имен.

Для того, чтобы получить правильное символьное имя можно пойти двумя способами:
1) Получить «искалеченное» имя функции с использованием RTTI:

```
typeid(func).name()
```

2) Отключить мэнглинг для данной функции:

```
extern "C" double func(std::string name);
```

Первый способ является плохо переносимым между компиляторами и требует подключения библиотеки `<typeinfo>`, что в свою очередь требует от компилятора включения в единицу трансляции информацию о типах времени выполнения (англ. *Runtime Type Information*, RTTI).
Второй способ более лаконичный — он использует правила компоновки для языка C, в котором ни классов, ни пространств имен, ни перегрузок функций нет, поэтому кодировать аргументы в имя символа не нужно, следовательно мэнглинг не выполняется.

7. Дополнительная информации

Для получения дополнительной информации по позднему связыванию с использованием `libdl.so` воспользуйтесь справочной информацией системного программиста POSIX:
https://man7.org/linux/man-pages/man0/dlfcn.h.0p.html
https://man7.org/linux/man-pages/man3/dlopen.3.html
https://man7.org/linux/man-pages/man3/dlsym.3.html