Лабораторна робота № 7

Тема. Статичні та динамічні бібліотеки. WINDOWS та LINUX

Мета. Ознайомитися з статичними та динамічними бібліотеками в операційних системах WINDOWS та LINUX. Навчитися реалізовувати статичні та динамічні бібліотеки.

Теоретичні відомості

Створення програмного забезпечення довгий і дорогий процес, тому існує необхідність його спростити та оптимізувати для подальшого модифікування застосунків. Часто виникає необхідність використання одних і тих ж функцій або даних у різних частинах проекту. Частина функцій у проекті використовується рідко, а частина при створені нової версії застосунку буде постійно змінюватись або додаватимуться нові функції. Крім того, всі дані можна розділити за певною тематикою або за видом ресурсів і т.д. Також потрібно передбачити оновлення даних. Тому виникає необхідність винести частину коду і даних з основної частин проекту в окремі приєднанні модулі.

Зрозуміло, що виносити кожну функцію або екземпляр даних в окремий файл чи модуль недоцільно, оскільки це спричинить довший відгук від програмного продукту, навантажить систему додатковими викликами, що у свою чергу зменшить продуктивність і ефективність застосунку. Зазвичай функції або дані об’єднують згідно функціональності, за тематикою та іншими властивостями в окремі скомпільовані спеціальні об’єктні файли – бібліотеки.

Бібліотека – це файл, що містить один або декілька об’єктних файлів для вирішення близьких за тематикою завдань у розробці програмних продуктів. У бібліотеці можуть міститися об’єктні модулі, програмний код або дані, що можуть використовуватись окремо або разом на різних етапах створення проекту (компілювання, лінкування, завантаження чи виконання).

Бібліотека містить символьний індекс, який складається з назв функцій і змінних, які містяться у бібліотеці. Це дозволяє прискорити процес лінкування програми, оскільки пошук функцій і змінних в об'єктних файлах бібліотеки відбувається набагато швидше, ніж пошук в наборі вказаних об'єктних файлів. Тому використання бібліотеки дозволяє компактно зберігати усі необхідні об'єктні файли в одному місці, і при цьому значно підвищити швидкість компіляції.

Бібліотеки прийнято розділяти відповідно до способу з’єднання на статичні та динамічні. Статичні бібліотеки зв’язуються з проектом перед завантаженням і є частиною бінарного файлу. Динамічні можуть бути зв’язані з проектом статично і динамічно. У різних операційних системах є свої особливості створення і використання бібліотек

WINDOWS

DLL (англ. Dynamic-link library — динамічно приєднувана бібліотека) — реалізовані компанією Microsoft загальні бібліотеки в ОС Windows та OS/2. Як правило бібліотеки мають розширення файлу *.dll, *.ocx (для бібліотек, що містять елементи керування ActiveX) або *.drv (драйвери старих версій ОС). DLL може містити код, дані та ресурси в будь-якій комбінації. Існує два способи зв’язати dll з кодом c++: статичний та динамічний. Для статичного зв’язування потрібно створити додатковий заготовочний файл, в якому кожну функцію потрібно експортувати з dll, а у заголовочному файлі самої програми потрібно імпортувати ці функції. Для того, щоб їх використовувати також потрібно підключити *.lib файл у проект де використовується dll. В динамічному зв’язуванні потрібно загрузити dll за допомогою фції LoadLibrary. Далі потрібно перевизначити тип для кожної функції. І присвоїти мінним типу, який ми визначили вказівник на ф-ції, який отримуємо за допомогою GetProcAddress. Практично неможливо створити додаток Windows, в якому невикористовувалися б бібліотеки DLL. У DLL містяться всі функції Win32 API і незліченну кількість інших функцій операційних систем Win32. Взагалі кажучи, DLL – це просто набори функцій, зібрані в бібліотеки. Однак, на відміну від своїх статичних родичів (файлів. Lib), бібліотеки DLL не приєднані безпосередньо до виконуваних файлів за допомогою редактора зв'язків. У виконуваний файл занесена тільки інформація про їхнє місце знаходження. У момент виконання програми завантажується вся бібліотека цілком. Завдяки цьому різні процеси можуть користуватися спільно одними і тими ж бібліотеками, знаходяться в пам'яті. Такий підхід дозволяє скоротити обсяг пам'яті, необхідний для декількох додатків, що використовують багато спільних бібліотек, а також контролювати розміри EXE-файлів. Однак, якщо бібліотека використовується тільки одним додатком, краще зробити її звичайною, статичною. Звичайно, якщо що входять до її складу функції будуть використовуватися тільки в одній програмі, можна просто вставити в неї відповідний файл з вихідним текстом. Найчастіше проект підключається до DLL статично, або неявно, на етапі компонування. Завантаженням DLL при виконанні програми управляє операційна система. Однак, DLL можна завантажити і явно, або динамічно, в ході роботи додатки.

Створення власної DLL З точки зору програміста - DLL являє собою бібліотеку функцій (ресурсів), якими може користуватися будь-який процес, що завантажив цю бібліотеку. Саме завантаження, до речі, забирає час і збільшує витрату споживаної додатком пам'яті; тому бездумне дроблення однієї програми на безліч DLL нічого доброго не принесе. Інша справа - якщо якісь функції використовуються декількома додатками. Тоді, помістивши їх в одну DLL, ми позбудемося дублювання коду і скоротимо загальний обсяг додатків - і на диску, і в оперативній пам'яті. Можна виносити в DLL і рідко використовувані функції окремого додатка; наприклад, деякі користувачі текстового редактора використовують в документах формули і діаграми - так навіщо ж відповідним функціям даремно "від'їдати" пам'ять? Завантаживши DLL процесу доступні не

всі її функції, а лише явно надаються самій DLL для "зовнішнього світу" - т. Н. Експортовані. Функції, призначені суто для "внутрішнього" користування, експортувати безглуздо (хоча і не заборонено). Чим більше функцій експортує DLL - тим повільніше вона завантажується; тому до проектування інтерфейсу (способу взаємодії DLL із зухвалим кодом) слід поставитися уважніше. Хороший інтерфейс інтуїтивно зрозумілий програмістові, небагатослівний і елегантний: як то кажуть, ні додати, ні відняти. Суворих рекомендацій на цей рахунок дати неможливо - вміння приходить з досвідом Для експортування функції з DLL - перед її описом слід вказати ключове слово __declspec (dllexport), як показано в наступному прикладі:

```
// Myfirstdll.c
#include <stdio.h>
 // Ключове слово __declspec (dllexport)
// Робить функцію експортованої
 __declspec (dllexport) void Demo (char * str) {
 // Виводимо на екран передану функції Demo рядок
 printf (str);
}
```

Для компіляції цього прикладу в режимі командного рядка можна запустити компілятор Microsoft Visual Studio: "cl.exe myfirstdll.c / LD". Ключ "/ LD" вказує линкерах, що потрібно отримати саме DLL. Для зборки DLL з інтегрованою оболонки Microsoft Visual Studio - при створенні нового проекту потрібно вибрати пункт "Win32 Dynamics Link Library", потім "An Empty DLL project"; потім перейти до закладки "File View" вікна "Workspace" - і, вибравши правою клавішею миші папку "Source Files", додати в проект новий файл ("Add Files to Folder"). Компіляція здійснюється як звичайно ("Build" ("Build"). Якщо все пройшло успішно - в поточній директорії (або в директорії Release \ Debug при компіляції з оболонки) з'явиться новий файл - "MyFirstDLL.dll". Давайте заглянемо в нього через "мікроскоп" - утиліту dumpbin, що входить в штатну поставку SDK і Microsoft Visual Studio: "dumpbin / EXPORTS MyFirstDLL.dll". Відповідь програми в кілька скорочено вигляді повинен виглядати так:

```
 Section contains the following exports for myfirst.dll
0 characteristics
0.00 version
1 ordinal base
1 number of functions
 1 number of names
1 0 00001000 Demo
```

Вийшло! Створена нами DLL дійсно експортує функцію "Demo" - залишається тільки розібратися, як її викликати Виклик функцій з DLL Існує два способи завантаження DLL: з явною і неявній компонуванням. При неявній компонуванні функції загружаемой DLL додаються в секцію імпорту файла, який викликає.

При запуску такого файлу завантажувач операційної системи аналізує секцію імпорту і підключає всі зазначені бібліотеки. Зважаючи на свою простоту цей спосіб користується великою популярністю; але простота - простотою, а неявній компонуванні притаманні певні недоліки та обмеження: 1. всі підключені DLL завантажуються завжди, навіть якщо протягом усього сеансу роботи програма ні разу не звернеться ні до однієї з них; 2. якщо хоча б одна із потрібних DLL відсутній (або DLL не експортує хоча б однієї необхідної функції) - завантаження виконуваного файлу переривається сполученням "Dynamic link library could not be found" (або щось в цьому роді) - навіть якщо відсутність цієї DLL некритично для виконання програми. Наприклад, текстовий редактор міг би цілком працювати і в мінімальній комплектації - без модуля друку, виведення таблиць, графіків, формул та інших другорядних компонентів, але якщо ці DLL завантажуються неявній компонуванням - хочеш не хочеш, доведеться "тягнути" їх за собою. 3. пошук DLL відбувається в наступному порядку: в каталозі, що містить викликає файл; в поточному каталозі процесу; в системному каталозі% Windows% System%; в основному каталозі% Windows%; в каталогах, зазначених у змінній PATH. Задати інший шлях пошуку неможливо (вірніше - можливо, але для цього буде потрібно вносити зміни до реєстру, і ці зміни вплинуть на всі процеси, реклама, яка в системі - що не є добре). Явне компонування усуває всі ці недоліки - ціною деякого ускладнення коду. Програмісту самому доведеться подбати про завантаження DLL і підключенні експортованих функцій (не забуваючи при цьому про контроль над помилками, інакше в один прекрасний момент справа скінчиться зависанням системи). Зате явна компоновка дозволяє довантажувати DLL в міру необхідності і дає програмісту можливість самостійно обробляти ситуації з відсутністю DLL. Можна піти й далі - не задавати ім'я DLL в програмі явно, а сканувати такий-то каталог на предмет наявності динамічних бібліотек і підключати всі знайдені до додатка. Саме так працює механізм підтримки plugin'ов в популярному файл-менеджері FAR (та й не тільки в ньому). Таким чином, неявній компонуванням доцільно користуватися лише для підключення завантажуваних в кожному сеансі, життєво необхідних для роботи додатку динамічних бібліотек; у всіх інших випадках - переважніше явна компоновка. Завантаження DLL з неявній компонуванням Щоб викликати функцію з DLL, її необхідно оголосити в зухвалій коді - або як external (т. Е. Як звичайну зовнішню функцію), або випередити ключовим словом __declspec (dllimport). Перший спосіб більш популярний, але другий все-таки переважніше - в цьому випадку компілятор, зрозумівши, що функція викликається саме з DLL, зможе відповідним чином оптимізувати код. Наприклад, функція "Demo" зі створеної нами бібліотеки - "MyFirstDll" викликається так:

```
// ImplictDll.c
// Оголошуємо зовнішню функцію Demo
__declspec (dllimport) void Demo (char * str);
```

```
 main () {
// Викликаємо функцію Demo з DLL
 Demo ("Hello, World! \ N");
}
```

З командного рядка даний приклад компілюється так: "cl.exe ImplictDll.c myfirstdll.lib", де "myfirstdll.lib" - ім'я бібліотеки, автоматично сформованої компоновщиком при створенні нашої DLL. Зрозуміло, "чужі" DLL не завжди поставляються разом з супутніми бібліотеками, але їх можна легко виготовити самостійно! На цей випадок передбачена спеціальна утиліта implib, яка постачається разом з компілятором, і викликається так: "implib.exe Ім'я_файлу _создаваемой_бібліотекі Імя_DLL". У нашому випадку - не будь у нас файлу "MyFirstDLL.lib", його довелося б отримати так: "implib.exe MyFirstDLL.lib MyFirstDLL.dll". З усіма стандартними DLL, що входять до складу Windows, цю операцію проробляти не потрібно, тому що необхідні бібліотеки поширюються разом із самим компілятором. Для підключення бібліотеки в інтегрованої середовищі Microsoft Visual Studio - в меню "Project" виберіть пункт "Project Settings", в діалоговому вікні перейдіть до закладки "Link" і допишіть ім'я бібліотеки в кінець рядка "Object / Library Modules", відокремивши її від решти символом пробілу. Якщо все пройшло успішно, з'явиться новий файл "ImplictDll.exe", який, будучи запущеним, гордовито виведе на екран "Hello, Word!". Це означає, що наша DLL підключена і успішно працює. Заглянемо всередину: як це відбувається? Запустимо "dumpbin / IMPORTS ImplictDll.exe" і подивимося, що нам повідомить програма:

```
File Type: EXECUTABLE IMAGE
Section contains the following imports:
myfirstdll.dll
404090 Import Address Table
4044C8 Import Name Table
0 time date stamp
0 Index of first forwarder reference
0 Demo
KERNEL32.dll
404000 Import Address Table
404438 Import Name Table
 0 time date stamp
0 Index of first forwarder reference
19B HeapCreate
2BF VirtualFree
CA GetCommandLineA
174 GetVersion
7D ExitProcess
29E TerminateProcess
```

F7 GetCurrentProcess

Ось вона - "Myfirstdll.dll", і ось функція "Demo", а крім неї - виявляє свою присутність бібліотека KERNEL32.DLL - вона необхідна RTL-коду (R un T ime L ibrary - бібліотека часу виконання), насильно вміщеної компілятором в наш додаток. RTL-код забезпечує роботу з динамічною пам'яттю (heap), зчитує аргументи командного рядка, перевіряє версію Windows і багато-багато іншого! Звідси і з'являються в таблиці імпорту функції HeapCreate, GetCommandLine, GetVersion і т.д. Так що - не дивуйтеся, побачивши "лівий" імпорт в своєму додатку! Простежити, як саме відбувається завантаження DLL, можна за допомогою відладчика. Загальновизнаний лідер - це, звичайно, SoftIce від NuMega, але для наших експериментів цілком зійде і штатний відладчик Microsoft Visual Studio. Відкомпільоване нашу зухвалу програму, натиснемо для покрокового прогону додатки Не встигло ще виконатися жодного рядка коду, як у вікні "output" відладчика з'явилися такі рядки, що свідчать про завантаження зовнішніх DLL: NTDLL.DLL, MyFirstDll.dll і Kernel32.dll. Так і має бути - при неявній компонуванні динамічні бібліотеки підключаються відразу ж при завантаженні файлу, задовго до виконання функції main!

Loaded 'C: \ WINNT \ System32 \ ntdll.dll', no matching symbolic information found.
Loaded 'F: \ ARTICLE \ PRG \ DLL.files \ myfirstdll.dll', no matching symbolic information found.
Loaded 'C: \ WINNT \ system32 \ kernel32.dll', no matching symbolic information found.

Завантаження DLL з явною компонуванням Явну завантаження динамічних бібліотек здійснює функція HINSTANCE LoadLibrary (LPCTSTR lpLibFileName) або її розширений аналог HINSTANCE LoadLibraryEx (LPCTSTR lpLibFileName, HANDLE hFile, DWORD dwFlags). Обидві вони експортуються з KERNEL32.DLL, отже, кожен додаток вимагає неявній компонування принаймні цієї бібліотеки. У разі успішного завантаження DLL повертається лінійний адресу бібліотеки в пам'яті. Передавши його функції FARPROC GetProcAddress (HMODULE hModule, LPCSTR lpProcName) - ми отримаємо покажчик на функцію lpProcName, експортовану даної DLL. При виникненні помилки обидві функції повертають NULL. Після завершення роботи з динамічною бібліотекою її слід звільнити викликом функції BOOL FreeLibrary (HMODULE hLibModule). Для пояснення приведемо код прикладу з докладними коментарями:

```
// DynCall.c
 #include <stdio.h>
 #include <windows.h>
 main () {
// Дескриптор загружаемой dll
HINSTANCE h;
// Оголошення покажчика на функцію, спричиненої з DLL
// Зверніть увагу - імена оголошуваною функції і
```

```
 // Функції, спричиненої з DLL, можуть і не співпадати,
 // Т.к. за вибір викликається функції відповідає
// GetProcAddress
 void (* DllFunc) (char * str);
// Завантажуємо MyFirstDLL
 h = LoadLibrary ("MyFirstDLL.dll");
// Контроль помилок - якщо завантаження пройшла успішно,
// Функція поверне щось відмінне від нуля
 if (! h)
 {
printf ("Помилка - не можу знайти MyFirstDLL.dll \ n");
 return;
}
 // Викликом GetProcAddress отримуємо адресу функції Demo
// І присвоюємо його вказівником DllFunc з явним
// Приведенням типів. Це необхідно тому що
// GetProcAddress повертає безтиповою far-покажчик
 DllFunc = (void (*) (char * str))GetProcAddress (h, "Demo");
// Контроль помилок - якщо виклик функції GetProcAddress
// Завершився успішно, вона поверне ненульовий покажчик
if (! DllFunc)
{
printf ("Помилка! У MyFirstDLL" "Відсутня ф-ція Demo \ n");
return;
}
// Виклик функції Demo з DLL
 DllFunc ("Test");
// Вивантаження динамічної бібліотеки з пам'яті
FreeLibrary (h);
}
```

Компілювати так: "cl DynCall.c" - ніяких додаткових бібліотек вказувати не потрібно (необхідні kernel32.lib і LIBC.lib компоновщик підключить самостійно). В інтегрованому середовищі Microsoft Visual Studio достатньо клацнути мишкою по іконці "Build" - і ніяких додаткових налаштувань! Для вивчення секції імпорту щойно отриманого файлу запустимо утиліту Dumpbin - зверніть увагу, що тут відсутня будь-яке згадування про MyFirstDLL.dll, але виявляються дві функції: LoadLibrary і GetProcAddress - які і завантажують нашу бібліотеку. Це дуже важлива обставина - вивчення секції імпорту досліджуваного файлу не завжди дозволяє встановити повний перелік функцій і динамічних бібліотек, які використовує додаток. Наявність LoadLibrary і GetProcAddress красномовно свідчить про те, що додаток підвантажує якісь модулі під час роботи самостійно.

Щоб з'ясувати які - запустимо його під отладчиком. Відразу ж після завантаження виконуваного файлу у вікні Output з'являться рядки:
Loaded 'C: \ WINNT \ System32 \ ntdll.dll', no matching symbolic information found.
Loaded 'C: \ WINNT \ system32 \ kernel32.dll', no matching symbolic information found.
Це знову вантажаться обов'язкові KERNEL32.DLL і NTDLL.DLL (останнє - тільки під Windows NT / 2000). Жодної згадки про MyFirstDLL.dll ще немає. Покроково виконуючи програму ("Debug" ("Step Over"), дочекаємося виконання функції LoadLibrary. Тут же в Output-вікні з'явиться наступна рядок:
Loaded 'F: \ ARTICLE \ PRG \ DLL.files \ myfirstdll.dll', no matching symbolic information found.
Наша динамічна бібліотека завантажилася; але не відразу після запуску файлу (як це відбувалося при неявній компонуванні), а тільки коли в ній виникла необхідність! Якщо ж з якихось причин DLL не знайдеться, або виявиться, що в ній відсутня функція Demo - операційна система не стане "вбивати" додаток з "некрологом" критичної помилки, а надасть програмісту можливість діяти самостійно. В якості експерименту спробуйте видалити (перейменувати) MyFirstDLL.dll і подивіться, що з цього вийде.
Вивантаження динамічних бібліотек з пам'яті
Коли завантажена динамічна бібліотека більше не потрібна - її можна звільнити, викликавши функцію BOOL FreeLibrary (HMODULE hLibModule) і передавши їй дескриптор бібліотеки, раніше повернутий функцією LoadLibrary. Зверніть увагу - DLL можна саме звільнити, але не вивантажити! Вивантаження DLL з пам'яті не гарантується, навіть якщо роботу з нею завершили всі раніше завантажили її процеси. Затримка вивантаження передбачена спеціально - на той випадок, якщо ця ж DLL через деякий час знову знадобиться якомусь процесу. Такий трюк оптимізує роботу часто використовуваних динамічних бібліотек, але погано підходить для рідко використовуваних DLL, що завантажуються лише одного разу на короткий час. Ніяких документованих способів насильно вивантажити динамічну бібліотеку з пам'яті ні; а ті, що є - працюють з ядром на низькому рівні і не можуть похвалитися переносимістю. Тому тут ми їх розглядати не будемо. До того ж - тактика звільнення і вивантаження DLL по-різному реалізована в кожній версії Windows: Microsoft, прагнучи підібрати найкращу стратегію, безперервно змінює цей алгоритм; а тому і відмовляється його документувати. Не можна не звернути уваги на одну дуже важливу обставину: динамічна бібліотека не володіє ніякими ресурсами - ними володіє, незалежно від способу компонування, що завантажив її процес. Динамічна бібліотека може відкривати файли, виділяти пам'ять і т. Д., Але пам'ять не буде автоматично звільнена після виклику FreeLibrary, а файли не опиняться самі собою закриті - все це відбудеться лише після завершення процесу, але не раніше! Природно, якщо програміст сам не звільнить всі непотрібні ресурси вручну, за допомогою функцій CloseHandle, FreeMemory і подібних їм. Якщо функція FreeLibrary

пропущена, DLL звільняється (але не факт, що вивантажується!) Тільки після завершення процесу, що викликав. Можуть виникнути сумніви: раз FreeLibrary негайно не вивантажує динамічну бібліотеку з пам'яті, так навіщо вона взагалі потрібна? Чи не краще тоді все пустити на самоплив - все одно ж завантажені DLL будуть гарантовано звільнені після завершення процесу? Що ж, доля правди тут є, і автор сам часом так і поступає; але при нестачі пам'яті операційна система може безперешкодно використовувати місце, зайняте звільненими динамічними бібліотеками під щось корисне - а якщо DLL ще не звільнені, їх доведеться "скидати" в файл підкачки, втрачаючи дорогоцінний час. Тому краще звільняйте DLL відразу ж після їх використання!

ООП і DLL

Динамічні бібліотеки нічого не знають ні про яке ООП! Вони не мають ні найменшого уявлення про існування класів! Все, що вміють DLL - експортувати одне або кілька імен функцій (ресурсів, глобальних змінних), а вже як його використовувати - вирішувати програмісту або компілятору. Що ж; випробуємо компілятор на "кмітливість", включивши в опис класу ключове слово __declspec (dllexport) - і подивимося, що з цього вийде:

```
 // DLLclass.cpp
#include <stdio.h>
class __declspec (dllexport) MyDllClass {
public:
Demo (char * str);
};
 MyDllClass :: Demo (char * str)
{ printf (str);
}
```

Відкомпілюємо цей код як звичайну DLL і заглянемо в таблицю імпорту утилітою dumpbin:

```
dumpbin / EXPORTS DLLclass.dll
File Type: DLL
Section contains the following exports for DLLclass.dll
0 characteristics
3B1B98E6 time date stamp Mon Jun 04 18:19:18 2001
0.00 version
1 ordinal base
2 number of functions
2 number of names
ordinal hint RVA name
1 0 00001000 ?? 4MyDllClass@@QAEAAV0 @ ABV0@@Z
2 1 00001020? Demo @ MyDllClass@@QAEHPAD @ Z
```

Таблиця імпорту явно не порожня - але виглядає дивно. Компілятор скалічив імена, щоб втиснути в них інформацію про класи і аргументах функцій без

порушень жорстких обмежень, що накладаються стандартом на символи, допустимі в експортованих іменах (наприклад, забороняється використовувати знак двокрапка, дужка і т. Д.). Як же з усім цим працювати? Спробуй-но вгадай - у що перетвориться те чи інше ім'я після компіляції! Втім, при неявній компонуванні ні про що гадати не доведеться, т. До. Про все подбає сам компілятор, а від програміста буде потрібно лише описати клас, випередивши його ключовим словом __declspec (dllimport):

```
// DLLClassCall.cpp
#include
class __declspec (dllimport) MyDllClass {
public: Demo (char * str); };
main () {
MyDllClass zzz;
zzz.Demo ("Привіт, ООП! А ми тут!");
}
```

Відкомпілюйте приклад як звичайну програму з неявній компонуванням ("cl DLLClassCall.cpp DLLClass.lib") і спробуйте запустити отриманий файл. Працює? Ніякої різниці з "класичним" Сі немає, чи не так? От тільки як підключити DLL з явною компонуванням? Невже не можна заборонити компілятору "калічити" імена функцій ?! Звичайно ж, можна

Мангла і як його побороти або імпорт класів з DLL явною компонуванням Спотворення функцій Си ++ компілятором називається по-англійськи "mangle", і серед програмістів широко поширена його калька - "Мангла". В принципі "манглеж" функцій не перешкоджає їх явного викликом допомогою GetProcAddress - достатньо лише знати, як точно називається та чи інша функція, що неважко з'ясувати тим же dumpbin. Однак засмічення програми подібними "завченими" іменами виглядає не дуже-то красиво; до того ж - всякий компілятор "Мангла" імена по-своєму, і безпосереднє використання імен призводить до непереносимості програми. Існує спосіб обійти спотворення імен - для цього необхідно підключити до линкерах спеціальний DEF-файл, капітуляційний імена, які не повинні змінитися. У нашому випадку він повинен виглядати так

```
// DllClass.def:
EXPORTS
Demo
```

Спершу йде ключове слово "EXPORTS", за яким слідують одне або декілька "недоторканних" імен. Кожне ім'я починається з нового рядка, і в його кінці не вказується крапка з комою.

Для підключення DEF-файлу при компіляції з командного рядка - використовуйте опцію "/ link /DEF:імя_файла.def", наприклад так: "cl DLLclass.cpp / LD / link /DEF:DLLClass.def". Для підключення DEF-файлу в інтегрованої середовищі Microsoft Visual Studio - перейдіть до закладки "File

View" вікна "Workspace" і, клацнувши правою клавішею по папці "Source Files", виберіть в контекстом меню пункт "Add Files to Folder", а потім вкажіть шлях до DEF-файлу. Відкомпілюйте проект як звичайно: "Build" ("Build". Зазирнувши в таблицю імпорту отриманого DLL-файлу, ми, серед іншої інформації, побачимо наступне:

1 0 00001000 ?? 4MyDllClass@@QAEAAV0 @ ABV0@@Z
2 1 00001020 Demo

Тепер ім'я функції Demo виглядає "як годиться". А абракадабра, розташована рядком вище - це конструктор класу MyDllClass, який, хоч і не був спеціально оголошений, все одно експортується з динамічної бібліотеки. Однак, позбувшись від однієї проблеми, ми отримуємо іншу - ім'я функції Demo втратило будь-яке уявлення про клас, якому воно належало; і тепер доведеться завантажувати його вручну, повторюючи цю операцію для кожного елемента класу. Фактично - доведеться в викликається програмі збирати "скелет" класу з "кісточок" заново. Але іншого способу явною завантаження класу з DLL не існує. Наступний приклад демонструє виклик функції MyDllClass з динамічної бібліотеки з явною компонуванням. Обробка помилок для спрощення опущена. Зверніть увагу, як оголошується функція Demo - опису класу в DLL і в викликає програмі істотно відрізняються, тому з ідеєю помістити опису класу в загальний для всіх include-файл доведеться розлучитися.

```
 // DeMangle.cpp
#include
 class MyDllClass {
public: void (* Demo) (char * str);
};
main () {
HINSTANCE h = LoadLibrary ("DllClass.dll");
MyDllClass zzz;
// Увага! Виконання конструктора / деструктора
 // Класу при явній завантаженні не відбувається
// Автоматично і при необхідності цю операцію
// Слід виконати вручну
zzz.Demo = (void (*) (char * str)) GetProcAddress (h, "Demo");
zzz.Demo ("Test"); }
```

Завантаження ресурсів з DLL

Крім функцій, динамічні бібліотеки можуть містити і ресурси - рядки, іконки, малюнки, діалоги і т. Д. Зберігання ресурсів в DLL дуже зручно; зокрема - при створенні додатків з багатомовним інтерфейсом: замінивши одну DLL на іншу, ми замінюємо всі написи в програмі, скажімо, з англійської на російську - і, зауважте, без всяких "хірургічних втручань" в код програми! Аналогічно можна міняти іконки, зовнішній вигляд діалогів і т. д. Створення DLL, що містить

ресурси, нічим не відрізняється від створення виконуваного додатки з ресурсами; спочатку необхідно створити сам файл ресурсів - наприклад, так:

```
// MyResDll.rc
#pragma code_page (1251)
STRINGTABLE DISCARDABLE
 BEGIN
1 "Hello, World!"
END
```

Файл ресурсів треба скомпілювати - "rc MyResDll.rc" - і перетворити лінкером в DLL, обов'язково вказавши прапор "/ NOENTRY", т. До. Ця динамічна бібліотека містить виключно одні ресурси і ні рядка коду: "link MyRedDll.res / DLL / NOENTRY ". В Visual Studio це зробити ще простіше - достатньо клікнути по папці "Resourses" вікна "File View" і додати новий файл ресурсу, який потім можна буде модифікувати візуальним редактором на свій розсуд. Для завантаження ресурсу з DLL - в принципі, можна скористатися вже знайомої нам функцією LoadLibray, і передавати возращением нею дескриптор LoadString або іншої функції, що працює з ресурсами. Однак завантаження динамічної бібліотеки можна значно прискорити, якщо "пояснити" системі, що ця DLL не містить нічого, крім ресурсів, і нам достатньо лише спроектувати її на адресний простір процесу, а про все інше ми зуміємо подбати і самостійно. Ось тут-то і стане в нагоді функція LoadLibraryEx: її перший аргумент, як і у колеги LoadLibrary, задає ім'я динамічної бібліотеки для завантаження, другий - зарезервований і має дорівнювати нулю, а третій, будучи рівним LOAD_LIBRARY_AS_DATAFILE, змушує функцію робити саме те, що нам потрібно - завантажувати DLL як базу даних (якщо динамічна бібліотека містить крім ресурсів ще й код, то завантаження з цим ключем проходить однаково успішно, але функції завантаженої DLL стануть недоступними - тільки ресурси):

```
// DllLoadRes.c
 #include
#include
 main () {
HINSTANCE h;
char buf [100];
h = LoadLibraryEx ("MyResDll.dll", 0, LOAD_LIBRARY_AS_DATAFILE);
LoadString (h, 1, & buf [0], 99);
 printf ("% s \ n", & buf [0]);
 }
```

Ця програма компілюються точно так само, як і попередні приклади явної компонування - і після запуску переможно виводить на екрані "Hello, Word!", Підтверджуючи, що ресурс "рядок" з динамічної бібліотеки був успішно завантажений! Аналогічним способом можна завантажувати ресурси з

виконуваних файлів; з цієї точки зору вони нічим не відрізняються від динамічних бібліотек

LINUX

Компіляція програм виконується командою: gcc <ім'я_файлу> або g++ <ім'я_файлу>

Після цього, якщо процес компіляції пройде успішно, то ви отримаєте завантажуваний файл a.out, запустити який можна командою: ./a.out

При створенні великих програм виникає необхідність розбити програму на частини які є логічно завершиними і функціонально обмеженими.

Для того, щоб винести функцію або змінну в окремий файл потрібно перед нею поставити зарезервоване слово extern.

Приклад файл main.c:

```
#include <stdio.h>
extern int f1();
int main()
{
int n1;
n1 = f1();
printf("f1() = %d\n",n1);
return 0;
}
```

Файл f1.c :

```
int f1()
{
return 2;
}
```

Компілювати можна усі файли одночасно однією командою, перераховуючи складені файли через пропуск після ключа -c: gcc -c main.c f1.c

Або кожен файл окремо:

```
gcc -c f1.c
gcc -c main.c
```

В результаті роботи компілятора ми отримаємо два окремі об'єктні файли:

main.o

f1.o

Щоб їх зібрати в один файл з допомогою gcc потрібно використати ключ -o, при цьому лінкер збере усі файли в один: gcc main.o f1.o -o rezult

В результаті виклику отриманої програми rezult командою: ./rezult

Таким чином можна створювати великі проекти.

В Linux є два види бібліотек: статичні та спільновикористованні(динамічні). Статичні бібліотеки приєднуються до коду програми на етапі компіляції. Динамічні завантажюються після запуску і їхнє лінкування відбувається на етапі

виконання. Є два методи використання спільновикористовуванних бібліотек: динамічне компанування в момент завантаження і динамічне заванаження з підключенням у програмі.

В простих програмах доцільніше використовувати статичні бібліотеки. В програмах які використовують декілька бібліотек, використання спільновикористовуванних дозволяє зменшити використання оперативної памяті під час виконання програми. Одна спільновикористовувана бібліотека може використовуватися одночасно декількома програмами при цьому в оперативній памяті вона знаходиться в єдиному екземплярі. У випадку статичної бібліотеки кожна програма завантажує свою власну копію бібліотечних функцій. При неявному завантаженні спільновикористовуваних бібліотек - динамічній компановці, завантаження бібліотеки при запуску програми бере на себе операційна система. У випадку явного - динамічного завантаження, програма явно завантажує бібліотеку і потім викликає потрібну функцію. Як приклад, так працює механізм завантаження програмних модулів - плагінів.

У Linux статичні бібліотеки зазвичай мають розширення . a (Archive), а спільно використовувані бібліотеки мають розширення . so (Shared Object). Зберігаються бібліотеки, як правило, в каталогах / lib і / usr / lib.

Для створення статичних бібліотек існує спеціальна проста програма називається ar (архіватор). Вона використовується для створення, модифікації і перегляду об'єктних файлів в статичних бібліотеках, які насправді представляють з себе прості архіви.

Приклад створимо з файлів f1.c і f2.c окрему бібліотеку. Скомпілюємо ці файли: gcc -c f1.c f2.c

Отримаємо два файли - f1.o і f2.o. Для того, щоб створити бібліотеку з обєктних файлів потрібно викликати програму ar з наступними параметрами: ar rc libім'я_бібліотеки.a [список_*.o_файлів]

Нехай бібліотека називатиметься fs, тоді: ar rc libfs.a f1.o f2.o

В результаті отримаємо файл libfs.a, у якому є копії обєктних файлів f1.o і f2.o. Якщо файл бібліотеки вже існує, то архіватор аналізуватиме вміст архіву, він додасть нові об'єктні файли і замінить старі оновленими версіями. Опція c примушує створювати (від create) бібліотеку, якщо її немає, а опція r (від replace) замінює старі об'єктні файли новими версіями.

Поки у нас є лише архівний файл libfs.a. Щоб з нього зробити повноцінну бібліотеку об'єктних файлів потрібно додати до цього архіву індекс символів, тобтосписок вкладених у бібліотеку функцій і змінних, щоб лінкування відбувалося швидше.: ranlib libім'я_бібліотеки.a

Програма ranlib додасть індекс до архіву і вийде повноцінна статична бібліотека об'єктних файлів. Варто відмітити, що на деяких системах програма ar автоматично створює індекс, і використання ranlib не має ніякого ефекту. Потрібно бути обережним при компіляції бібліотеки за допомогою файлів makefile, якщо ви не використовуватимете утиліту ranlib, то можливо на

якихось системах бібліотеки створюватимуться не вірно і загубиться незалежність від платформи.
Для компіляції нашого основного файлу main.c потрібно повідомити компілятор, що потрібно використати бібліотеки. Щоб компілятор знав де шукати бібліотеки йому потрібно повідомити каталог, в якому вони містяться і список цих билиотек. Каталог з бібліотеками вказується ключем -L, у нашому випадку бібліотека знаходиться в поточному каталозі, означає шлях до неї буде у вигляді точки (-L.). Використовувані бібліотеки перераховуються через ключ -l, після якого вказується назва бібліотеки без префікса lib і закінчення .a. У нашому випадку цей ключ виглядатиме, як -lfs. Тепер все однією командою:
gcc -c main.c
gcc main.o -L. -lfs -o rezult
Або: gcc main.c -L. -lfs -o rezult
Спершу варто сказати, що об'єктний файл створюваний нашим перевіреним способом зовсім не підходить для динамічних бібліотек.Пов'язано це з тим, що усіх об'єктних файлів створювані звичайним способом не мають уявлення про те в які адреси пам'яті буде завантажена програма, що використовує їх. Декілька різних програм можуть використати одну бібліотеку, і кожна з них розташовується в різному адресному просторі. Тому вимагається, щоб переходи у функції бібліотеки(операції goto на асемблері) використали не абсолютну адресацію, а відносну.Тобто генерований компілятором код має бути незалежним від адрес, така технологія дістала назву PIC - Position Independent Code. У компіляторі gcc ця можливість включається ключем -fPIC.
Тепер компіляція наших файлів матиме вигляд:
dron:~# gcc -fPIC -c f1.c
dron:~# gcc -fPIC -c f2.c
Динамічна бібліотека це вже не архівний файл, а справжня завантажувана програма, тому створенням динамічних бібліотек займається сам компілятор gcc. Для того, щоб створити динамічну бібліотеку потрібно використати ключ -shared:
dron:~# gcc -shared -o libfsdyn.so f1.o f2.o
В результаті отримаємо динамічну бібліотеку libfsdyn.so, яка по моїй задумці буде динамічною версією бібліотеки libfs.a, що видно з назви :) Тепер, щоб компілювати результуючий файл з використанням динамічної бібліотеки нам потрібно зібрати файл командою:
dron:~# gcc -з main.з
dron:~# gcc main.o -L. -lfsdyn -o rezultdyn
Якщо тепер Ви порівняєте файли, отримані при використанні статичної і динамічної бібліотеки, то побачите, що їх розміри відрізняються.В даному випадку файл створений з динамічною бібліотекою займає трохи більше місця, але це лише від того, що програма використовувана нами абсолютно примітивна і левову частку там займає спеціальний код для використання динамічних

можливостей. У реальних умовах, коли використовуються дуже великі функції розмір програми з використанням динамічної бібліотеки значно менший.
На цьому фокуси не кінчаються, якщо Ви зараз спробуєте запустити файл rezultdyn, те отримаєте помилку:
dron:~# ./rezultdyn
./rezultdyn: error in loading shared libraries: libfsdyn.so: cannot open
shared object file: No such file or directory
dron:~#
Це повідомлення видає динамічний линковщик. Він просто не може знайти файл нашої динамічної бібліотеки. Річ у тому, що завантажувач шукає файли динамічних бібліотек у відомих йому директоріях, а наша директорія йому не відома. Але це ми трохи відкладемо, тому що це досить складне питання.
А зараз варто поговорити ще про один момент використання бібліотек. Я спеціально створив динамічну бібліотеку з назвою fsdyn, щоб вона відрізнялася від назви статичної бібліотеки fs. Річ у тому, що якщо у Вас дві бібліотеки статична і динамічна з однаковими назвами, тобто libfs.a і libfs.so, те компілятор завжди використовуватиме динамічну бібліотеку.
Пов'язано це з тим, що в ключі -l задається частина імені бібліотеки, а префікс lib і закінчення .a чи .so приставляє сам компілятор. Так от алгоритм роботи компілятора такий, що якщо є динамічна бібліотека, то вона використовується за умовчанням.Статична ж бібліотека використовується коли компілятор не може виявити файл .so цієї бібліотеки. У усій наявній у мене документації пишеться, що якщо використати ключ -static, те можна насильно змусити компілятор використати статичну бібліотеку. Відмінно, спробуємо...
dron:~# gcc -staticmain.o -L. -lfs -o rez1
Як би я не пробував грати з позицією ключа -static, результуючий файл rez1 виходить розміром в 900 Кб. Після застосування програми strip розмір її зменшується до 200 Кб, але це ж не порівняти з тим, що наша перша статична компіляція давала програму розміром 10 Кб. А пов'язано це з тим, що будь-яка програма написана на C/C++ у Linux використовує стандартну бібліотеку "C" library, яка містить в собі визначення таких функцій, як printf(), write() і усіх інших. Ця бібліотека линкується до файлу як динамічна, щоб усі програми написані на C++ могли використати одного разу завантажені функції. Ну, а при вказівці ключа -static компілятор робить линковку libc статичною, тому розмір коду збільшується на усі 200 Кб.
Цей ефект я не зміг побороти. Тому я дійшов висновку, що статичну бібліотеку і динамічну краще всього створювати з різними іменами. Сподіваюся в майбутньому ми з Вами ще розберемося з цією заковикою, але якщо дійсно все так погано, то рішення з різними іменами буде єдино вірним.
У минулий раз ми з Вами виявили, що запуск програм, скомпільованих разом з динамічними бібліотеками, викликає помилку:

dron:~# ./rezultdyn
./rezultdyn: error in loading shared libraries: libfsdyn.so: cannot open
shared object file: No such file or directorydron:/#
Це повідомлення видає завантажувач динамічних бібліотек(динамічний линковщик - dynamic linker), який в нашому випадку не може виявити бібліотеку libfsdyn.so. Для налаштування динамічного линковщика існує ряд програм.
Перша програма називається ldd. Вона видає на екран список динамічних бібліотек використовуваних в програмі і їх місце розташування. В якості параметра їй повідомляється назва обстежуваної програми. Давайте спробуємо використати її для нашої програми rezultdyn:
dron:~# ldd rezultdyn
    libfsdyn.so => not found
    libc.so.6 => /lib/libc.so.6 (0x40016000)
    /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
dron:~#
Як бачите все правильно. Програма використовує три бібліотеки:
libc.so.6 - стандартну бібліотеку функцій мови C++.
ld-linux.so.2 - бібліотеку динамічної линковки програм ELF формату.
libfsdyn.so - нашу динамічну бібліотеку функцій.
Нашу бібліотеку вона знайти не може.І правильно! Динамічний линковщик шукає бібліотеки тільки у відомих йому каталогах, а каталог нашої програми йому явно не відомий.
Для того, щоб додати нашу директорію з бібліотекою в список відомих директорій потрібно подредактировать файл /etc/ld.so.conf. Наприклад, у мене цей файл складається з таких рядків:
dron:~# cat /etc/ld.so.conf
/usr/X11R6/lib
/usr/i386-slackware-linux/lib
/usr/i386-slackware-linux-gnulibc1/lib
/usr/i386-slackware-linux-gnuaout/lib
dron:~#
У усіх цих директорії зберігаються усіма використовувані бібліотеки. У цьому списку немає лише однієї директорії - /lib, яка сама по собі не потребує опису, оскільки вона є головною.Виходить, що наша бібліотека стане "помітною", якщо помістити її в один їх цих каталогів, або окремо описати в окремому каталозі. Давайте для тесту опишемо, додамо рядок в кінець файлу ld.so.conf:
/root
У мене цей файл валяється в домашньому каталогу користувача root, у Вас він може бути у іншому місці. Тепер після цього динамічний линковщик знатиме де можна знайти наш файл, але після зміни конфігураційного

файлу ld.so.conf необхідно, щоб система перечитала налаштування наново.Це робить програма ldconfig. Пробуємо запустити нашу програму:

```
dron:~# ldconfig
dron:~# ./rezultdyn
f1() = 25
f2() = 10
dron:~#
```

Як бачите все запрацювало :) Якщо тепер Ви видалите доданий нами рядок і знову запустите ldconfig, те дані про розташування нашої бібліотеки зникнуть і з'являтиметься таже сама помилка.

Але описаний метод впливає на усю систему в цілому і вимагає доступу адміністратора системи, тобто root. А якщо Ви простий користувач без понад можливості ?!

Для такого випадку є інше безболісне рішення.Це використання спеціального змінного середовища LD_LIBRARY_PATH, у якій перераховуються усі каталоги призначені для користувача динамічні бібліотеки, що містять. Для того, щоб встановити цю змінну в командному середовищі bash потрібно набрати всього декілька команд. Спершу подивимося чи є у нас така змінна середовища :

```
dron:~# echo $LD_LIBRARY_PATH
```

У мене у відповідь виводиться порожній рядок, що означає, що такого змінного середовища немає. Встановлюється вона таким чином:

```
dron:~# LD_LIBRARY_PATH=/root
dron:~# export LD_LIBRARY_PATH
```

Після цього програма rezultdyn прекрасно працюватиме.У разі, якщо у Вас в системі ця змінна середовища вже уставновлена, то, щоб не зіпсувати її значення, потрібно новий каталог додати до старого значення. Робиться це іншою командою:

```
dron:~# LD_LIBRARY_PATH=/root:${LD_LIBRARY_PATH}
dron:~# export LD_LIBRARY_PATH
```

Якщо Ви обнулите цю змінну, то знову бібліотека перестане працювати:

```
dron:~# LD_LIBRARY_PATH=""
dron:~# export LD_LIBRARY_PATH
dron:~# ./rezultdyn
./rezultdyn: error in loading shared libraries: libfsdyn.so: cannot open
shared object file: No such file or directory
dron:~#
```

Ви також паралельно можете зайти в систему під іншим користувачем або навіть тим же самим, але якщо Ви захочете проглянути значення LD_LIBRARY_PATH, те побачите її колишнє значення. Це означає, що два різні користувачі Linux не можуть впливати на роботу один одного, а це і є найголовніше хороша відмінність систем Unix від більшості інших систем

Якщо Ви подумали, що фокуси з динамічними бібліотеками закінчилися, то Ви дуже сильно помилилися. До того були квіточки, а ягідки будуть зараз :)
Виявляється, що використати динамічні бібліотеки можна не лише на початку завантаження, але і в процесі самої роботи програми. Програма сама може викликати будь-які функції з бібліотеки, коли їй захочеться.Для цього всього-лише потрібно використати бібліотеку dl, яка дозволяє линковать бібліотеки "на льоту". Вона управляє завантаженням динамічних бібліотек, викликом функцій з них і вивантаженням після кінця роботи.
Для використання функцій програмної роботи з динамічними бібліотеками необхідно підключити заголовний файл:
#include <dlfcn.h>
Щоб викликати якісь функції з динамічної бібліотеки спочатку потрібно відкрити цю бібліотеку(можна сказати "завантажити"). Відкривається вона функцією:
void *dlopen (const char *filename, int flag);
Параметр filename містить шлях до необхідної бібліотеки, а параметр flag задає деякі специфічні прапори для роботи. Функція повертає покажчик на завантажену бібліотеку. У разі будь-якої помилки повертається покажчик NULL. У такому разі тест помилки зрозумілий людині можна отримати за допомогою функції dlerror(). Поки ми не замислюватимемося над цим, і я приведу стандартний код для відкриття бібліотеки :

```
void *library_handler;
//.....
//завантаження бібліотеки

library_handler = dlopen("/path/to/the/library.so",RTLD_LAZY);
if (!library_handler){
        //якщо помилка, то вивести її на екран
        fprintf(stderr,"dlopen() error: %s\n", dlerror());
        exit(1); // у разі помилки можна, наприклад, закінчити роботу програми
};
```

Після цього можна працювати з бібліотекою. А робота ця полягає в отриманні адреси необхідної функції з бібліотеки. Отримати адресу функції або змінної можна по її імені за допомогою функції:
void *dlsym(void *handle, char *symbol);
Для цієї функції потрібно адресу завантаженої бібліотеки handle, отриманий при відкритті функцією dlopen(). Необхідна функція або змінна задається своїм ім'ям в змінній symbol.
Закривається бібліотека функцією:
dlclose(void *handle);
При закритті бібліотеки динамічний линковщик перевіряє лічильник кількості відкриттів бібліотеки, і якщо вона була відкрита декількома програмами

одночасно, то вона не вивантажується до тих пір, поки усі програми не закриють цю бібліотеку.
Для прикладу створимо програму, яка в якості параметра отримує назву функції, яку вона використовуватиме в роботі. Наприклад, це будуть математичні функції піднесення до степеня. Створимо спочатку динамічну бібліотеку.Пишемо її код:

```
double power2(double x){
	return x*x;
};

double power3(double x){
	return x*x*x;
};

double power4(double x){
	return power2(x)*power2(x);
};
//.....
```

Зберігаємо його у файл lib.c і створюємо динамічну бібліотеку libpowers.so наступними командами:

```
dron:~# gcc -fPIC -c lib.c
dron:~# gcc -shared lib.o -o libpowers.so
```

Тепер створюємо основну програму у файлі main.c:

```
#include <stdio.h>
/* заголовний файл для роботи з динамічними бібліотеками */
#include <dlfcn.h>

int main(int argc, char* argv[]){

	void *ext_library;  // хандлер зовнішньої бібліотеки
	double value=0;		// значення для тесту
	double (*powerfunc)(double x);// змінна для зберігання адреси функції

	//завантаження бібліотеки
	ext_library = dlopen("/root/libpowers.so",RTLD_LAZY);
	if (!ext_library){
		//якщо помилка, то вивести її на екран
		fprintf(stderr,"dlopen() error: %s\n", dlerror());
		return 1;
	};

	//завантажуємо з бібліотеки необхідну процедуру
	powerfunc = dlsym(ext_library, argv[1]);
```

```
        value=3.0;

        //виводимо результат роботи процедури
        printf("%s(%f) = %f\n",argv[1],value,(*powerfunc)(value));

        //закриваємо бібліотеку
        dlclose(ext_library);
};
```

Код головної програми готовий. Вимагається його відкомпілювати з використанням бібліотеки dl:

```
dron:~# gcc main.c -o main -ldl
```

Отримаємо програмний файл main, який можна тестувати. Наша програма повинна зводити значення 3.0 у потрібну нами міру, яка задається назвою функції. Давайте спробуємо:

```
dron:~# ./main power2
power2(3.000000) = 9.000000
dron:~# ./main power3
power3(3.000000) = 27.000000
dron:~# ./main power4
power4(3.000000) = 81.000000
dron:~#
```

Ну, як ?! Круто !!! Ми використовуємо якісь функції, знаючи лише їх назву.Представте можливості, що відкриваються, для програм, на основі цього методу можна створювати плагіни для програм, модернізувати якісь частини, додавати нові можливості і багато що інше.

.Досі ми писали з Вами прості динамічні бібліотеки, тому у нас не хворіла голова про ініціалізацію внутрішніх змінних. А уявіть собі складнішу ситуацію, коли функції бібліотеки для роботи вимагають змінні, що правильно ініціалізували. Ну, наприклад, для роботи функції потрібний буфер або масив.

Спеціально для таких випадків у бібліотеках можна задавати ту, що ініціалізувала і деинициализирующую функції:

```
void _init() - ініціалізація
void _fini() - деинициализация
```

Щоб зрозуміти, що до чого, введемо в нашій бібліотеці lib.c змінну test і функцію, що повертає її :

```
char *test;

char *ret_test(){
        return test;
};
```

Пишемо основну програму main.c. Вона дуже схожа на попередній наш проект, тому можете його модифікувати:

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

int main(){

	void *ext_library;
	double value=0;
	char * (*ret_test)();

	ext_library = dlopen("libtest.so",RTLD_LAZY);
	if (!ext_library){
		fprintf(stderr,"dlopen() error: %s\n", dlerror());
		return 1;
	};

	ret_test = dlsym(ext_library,"ret_test");

	printf("Return of ret_test: \"%s\" [%p]\n",(*ret_test)(),(*ret_test)());

	dlclose(ext_library);
};
```

Після компіляції усього цього господарства ми отримаємо результат:

```
dron:~# gcc -c lib.c -fPIC
dron:~# gcc -shared lib.o -o libtest.so
dron:~# gcc -o main main.c -ldl
dron:~# ./main
Return of ret_test: "(null)" [(nil)]
dron:~#
```

Як бачите змінна test виявилася рівною NULL, а нам би хотілося щось інше.Ну, так давайте подивимося як працюють функції _init() і _fini(). Створимо другу бібліотеку lib1.c:

```
#include <stdlib.h>

char *test;

char *ret_test(){
	return test;
};

void _init(){
	test=(char *)malloc(6);
	if (test!=NULL){
```

```
                *(test+0)='d';
                *(test+1)='r';
                *(test+2)='o';
                *(test+3)='n';
                *(test+4)='!';
                *(test+5)=0;
        };
        printf("_init() executed...\n");
};

void _fini(){
        if (test!=NULL) free(test);
        printf("_fini() executed...\n");
};
```

Тепер пробуємо компілювати:

```
dron:~# gcc -c lib1.c -fPIC
dron:~# gcc -shared lib1.o -o libtest.so
lib1.o: In function `_init':
lib1.o(.text+0x24): multiple definition of `_init'
/usr/lib/crti.o(.init+0x0): first defined here
lib1.o: In function `_fini':
lib1.o(.text+0xc0): multiple definition of `_fini'
/usr/lib/crti.o(.fini+0x0): first defined here
collect2: ld returned 1 exit status
dron:~#
```

Опаньки... Облом. Що ж це таке ?! Виявляється хтось вже використав ці функції до нас і програма не може слинковаться. Після довгого копання в декількох чужих первинниках я отримав відповідь на це питання. Виявляється, щоб позбавитися від бібліотеки, що заважає, потрібно використати ключ компілятора -nostdlib. Спробуємо:

```
dron:~# gcc -shared -nostdlib lib1.o -o libtest.so
dron:~#
```

Дивіться, все прекрасно скомилировалось.Тепер спробуємо запустити main:

```
dron:~# ./main
_init() executed...
Return of ret_test: "dron!" [0x8049c20]
_fini() executed...
dron:~#
```

Ну як ? Помоему класно. Тепер можна спокійно створювати для роботи бібліотеки змінні, що правильно ініціалізували. Проте класні ці штуки - динамічні бібліотеки ! А Ви що хотіли ? Той же Windows тільки на своїх DLL і живе. А Linux нічим не гірше...

lib.cpp

```
int MyApp() {
  return 0;
}
```

main.cpp

```
extern "C" int MyApp();
int main() {
  return MyApp();
}
```

Команди для компіляції статичної бібліотеки libstat.a

```
g++ -c lib.cpp
ar rc libstat.a lib.o
ranlib libstat.a
ar -t libstat.a
g++ -c main.cpp
g++ main.o -lpthread -lrt -L. -lstat -o statprog
```

Команди для компіляції динамічної бібліотеки libdyn.so та статичного зв’язування

```
g++ -fPIC -c lib.cpp -o ./stat/lib.o
g++ -shared -o libdyn.so ./stat/lib.o
g++ -c main.cpp -o ./stat/main.o
g++ ./stat/main.o -lpthread -lrt -L. -ldyn -o dynprog
LD_LIBRARY_PATH=/home/user/lab5os/so:${LD_LIBRARY_PATH}
export LD_LIBRARY_PATH
echo $LD_LIBRARY_PATH
```

main_dynamic_linc.cpp

```
#include <dlfcn.h>
#include <stdio.h>
int main() {
  void *MyLib;
  typedef int(*func)(void);
  func MyApp;
  MyLib = dlopen("./libdyn.so", RTLD_LAZY);
  if(MyLib == 0)
    return -1;
  else
```

```
    printf("libdyn.so is loaded!\n");
  MyApp = (func)dlsym(MyLib, "MyApp");
  (*MyApp)();
  printf("MyApp function is finished!\n");
  dlclose(MyLib);
  printf("libdyn.so is closed!\n");
  return 0;
}
```

Команди для компіляції програми з динамічним зв’язуванням

```
g++ -c main_dyn_linc.cpp -o ./dyn/main.o
g++ ./dyn/main.o -lpthread -lrt -ldl -o dynprog_dyn_linc
chmod +x dynprog_dyn_linc
./dynprog_dyn_linc
```

Завдання.

1. Реалізувати лабораторну роботу №5 (згідно варіанту) у вигляді статичної та динамічної бібліотеки в ОС WINDOWS.
2. Запустити створену динамічну бібліотеку з командної стрічки (cmd.exe) за допомогою rundll32.exe.
3. Створити окрему програму і реалізувати статичний зв'язок між програмою та бібліотекою із п. 1.
4. Створити окрему програму і реалізувати динамічний зв'язок між програмою та бібліотекою із п. 1.
5. Експортувати головну функцію бібліотеки під іншим іменем із п. 1.
6. Реалізувати лабораторну роботу №6 у вигляді статичної та динамічної (поділюваної) бібліотеки в ОС LINUX.
7. Створити окрему програму і реалізувати статичний зв'язок між програмою та бібліотекою із п. 2
8. Створити окрему програму і реалізувати динамічний зв'язок між програмою та бібліотекою із п. 2.
9. Порівняти результати виконання програми та роботу бібліотек під ОС Windows та Linux.
10. Результати виконання роботи відобразити у звіті.