# Лабораторная работа №2

## **Задание 1:**

### Необходимые знания

Как менять занчения переменных внутри функций в С

Для изменения значения параметра функция должна знать адрес памяти параметра. Чтобы сообщить функции адрес параметра, ваши программы должны использовать оператор адреса C++ (&). Следующий вызов функции иллюстрирует, как программа будет использовать оператор адреса для передачи адресов переменных *big* и *small* в функцию *change\_values:*

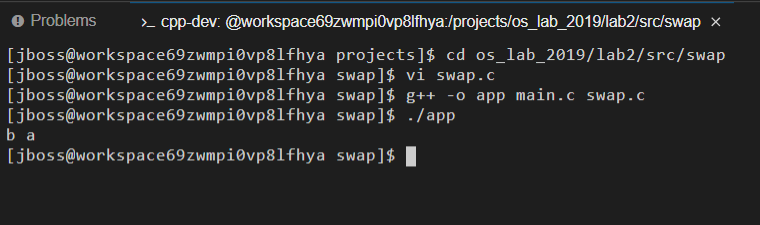
change\_values (&big, &small); ***-->Передача параметров по адресу***

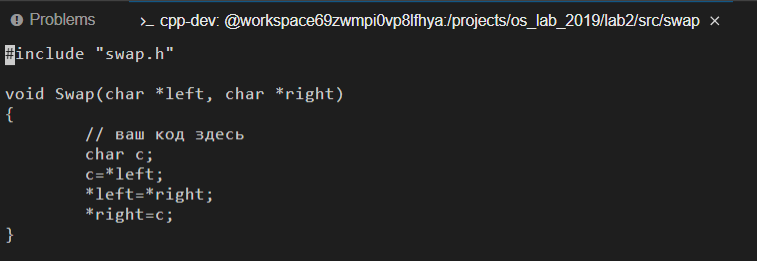
Внутри функции вы должны сообщить C++ , что программа будет передавать параметры с помощью адреса. Для этого вы объявляете *переменные-указатели,*предваряя имя каждой переменной звездочкой, как показано ниже:

void сhange\_values (int. \*big, int. \*small)***---> Указатель на тип int***

Переменная-указатель представляет собой переменную, которая содержит адрес памяти. Внутри функции вы должны сообщить C++ , что функция работает с адресом параметра

Задание





## **Задание 2:**

### Необходимые знания

1)Выделение и освобождение памяти в куче Си.

Очень часто возникают задачи обработки массивов данных, размерность которых заранее неизвестна. В этом случае возможно использование одного из двух подходов:

* + выделение памяти под [статический массив](https://prog-cpp.ru/c-massiv/), содержащий максимально возможное число элементов, однако в этом случае память расходуется не рационально;
  + динамическое выделение памяти для хранение массива данных.

Для использования функций динамического выделения памяти необходимо описать [указатель](https://prog-cpp.ru/c-pointers/), представляющий собой начальный адрес хранения элементов массива.

int \*p; // указатель на тип int

Начальный адрес статического массива определяется компилятором в момент его объявления и не может быть изменен.  
  
Для динамического массива начальный адрес присваивается объявленному указателю на массив в процессе выполнения программы.

**Стандартные функции динамического выделения памяти**

Функции динамического выделения памяти находят в оперативной памяти непрерывный участок требуемой длины и возвращают начальный адрес этого участка.  
  
Функции динамического распределения памяти:

void\* malloc(РазмерМассиваВБайтах);  
void\* calloc(ЧислоЭлементов, РазмерЭлементаВБайтах);

Для использования функций динамического распределения памяти необходимо подключение библиотеки <malloc.h>:

#include <malloc.h>

Поскольку обе представленные функции в качестве возвращаемого значения имеют указатель на пустой тип void, требуется явное приведение типа возвращаемого значения.  
  
Для определения размера массива в байтах, используемого в качестве аргумента функции malloc() требуется количество элементов умножить на размер одного элемента. Поскольку элементами массива могут быть как данные простых типов, так и составных типов (например, структуры), для точного определения размера элемента в общем случае рекомендуется использование функции

int sizeof(тип);

которая определяет количество байт, занимаемое элементом указанного типа.

Память, динамически выделенная с использованием функций calloc(), malloc(), может быть освобождена с использованием функции

free(указатель);

**Перераспределение памяти**

Если размер выделяемой памяти нельзя задать заранее, например при вводе последовательности значений до определенной команды, то для увеличения размера массива при вводе следующего значения необходимо выполнить следующие действия:

* Выделить блок памяти размерности n+1 (на 1 больше текущего размера массива)
* Скопировать все значения, хранящиеся в массиве во вновь выделенную область памяти
* Освободить память, выделенную ранее для хранения массива
* Переместить указатель начала массива на начало вновь выделенной области памяти
* Дополнить массив последним введенным значением

Все перечисленные выше действия (кроме последнего) выполняет функция

 void\* realloc (void\* ptr, size\_t size);

* ptr - указатель на блок ранее выделенной памяти функциями malloc(), calloc() или realloc() для перемещения в новое место. Если этот параметр равен NULL, то выделяется новый блок, и функция возвращает на него указатель.
* size - новый размер, в байтах, выделяемого блока памяти. Если size = 0, ранее выделенная память освобождается и функция возвращает нулевой указатель, ptr устанавливается в NULL.

Размер блока памяти, на который ссылается параметр ptr изменяется на size байтов. Блок памяти может уменьшаться или увеличиваться в размере. Содержимое блока памяти сохраняется даже если новый блок имеет меньший размер, чем старый. Но отбрасываются те данные, которые выходят за рамки нового блока. Если новый блок памяти больше старого, то содержимое вновь выделенной памяти будет неопределенным.

В чем разница между стеком и кучей .

Stack и Heap - это сегменты памяти, используемые в методах выделения памяти. Основное различие между стеком и кучей заключается в том, что стек включает в себя линейное и последовательное выделение памяти, которая используется для статического выделения памяти, тогда как куча выступает в качестве пула области хранения, которая выделяет память случайным образом (динамическое выделение памяти).

Скорость является основным параметром, который различает стек и кучу; стек значительно быстрее, чем куча.

## Сравнительная таблица

|  |
| --- |
|  |
| **Основа для сравнения** | **стек** | **отвал** |
| основной | Память выделяется в (LIFO) режиме «последним первым делом». | Память выделяется в случайном порядке. |
| Распределение и выделение | автоматическая | Руководство по эксплуатации |
| Стоимость | Меньше | Больше |
| Реализация | Жесткий | Легко |
| Вызов | НА) | O (1) |
| вопрос | Нехватка памяти | Фрагментация памяти |
| Местонахождение ссылки | Превосходно | адекватный |
| гибкость | Фиксированный размер и не гибкий | Изменение размера возможно |
| Время доступа | Быстрее | Помедленнее |

## Определение стека

Распределение стеков следует стратегии LIFO («Последний пришел - первым вышел») для выделения памяти процессам с помощью операций push и pop. Каждый блок в памяти имеет фиксированный размер, который не может быть расширен или сокращен. Последняя запись в стеке доступна в любой момент. Стек использует непрерывную память, где указатель, названный в качестве базового стека, указывает на первую запись стека, а другой указатель, названный как верхняя часть стека, указывает на последнюю запись стека.

Стек также поддерживает вызовы функций. Вызов функции может содержать коллекцию записей стека, известную как кадр стека. Другое имя стекового фрейма - запись активации в con компилятора, поскольку в нем хранятся данные, используемые во время компиляции программы. Всякий раз, когда вызывается функция, кадр стека помещается в стек.

Фрейм стека состоит из адресов или значений параметра функции и адреса возврата, который указывает, куда должен быть возвращен элемент управления после завершения выполнения функции.

Определение кучи

Распределение кучи не следует определенному подходу; скорее это позволяет случайное назначение и удаление памяти. Запрос присваивания процессом возвращает указатель на выделенную область памяти в куче, и процесс получает доступ к выделенной области памяти через указатель.

Выделение осуществляется через запрос на освобождение, в отличие от стека, где память освобождается автоматически. Куча создает дыры в распределении памяти при построении и освобождении структур данных. Он используется во время выполнения.

1. В стеке распределение и освобождение выполняются центральным процессором и выполняется автоматически, тогда как в куче программист должен делать это вручную.
2. Обработка кадров кучи обходится дороже, чем обработка кадров стека.
3. Реализация стека сложна. В отличие от этого, реализация кучи проста.
4. Вызов функции в стеке занимает O (N) времени. В отличие от этого, требуется куча O (1) времени.
5. Реализация стека в основном страдает от нехватки памяти. Напротив, основной проблемой в куче является фрагментация.
6. Доступ к кадру стека проще, чем кучи, поскольку стек ограничен небольшой областью памяти и всегда попадает в кэш, но кадры кучи разбросаны по всей памяти, поэтому доступ к памяти может привести к большему количеству пропусков кеша.
7. Стек не гибкий, выделенный объем памяти изменить нельзя. С другой стороны, куча является гибкой, и выделенная память может быть изменена.
8. Куча требует больше времени доступа, чем стек.

## Вывод

Распределение стека происходит быстрее, но сложнее. С другой стороны, куча медленнее, но ее реализация проще, чем стек. Куча более эффективна, чем стек.

Использование аргументов командной строки

Самый простой способ обработки аргументов командной строки – установка флага с помощью условного оператора:

if [ "$1" = "-a" ]; then flag\_a=1 fi

При реализации такого подхода возникает несколько проблем. Одна из них – захламление исходного кода дополнительными конструкциями. Так, перед этим примером необходимо предварительно обнулить переменную с помощью выражения **flag\_a=0**. Иначе нельзя точно определить, какое значение присвоит ей оболочка командной строки при инициализации.

Другая проблема состоит в том, что выполнение этого блока никак не влияет на параметры командной строки: **$1** по-прежнему может быть флагом (**-a**), другим аргументом или значением, введённым пользователем. Следовательно необходимо будет добавить дополнительные проверки при считывании остальных опций.

## Обработка нескольких аргументов командной строки в shell-скрипте

Для наглядности представим, что существует простой скрипт. Он работает в качестве обёртки к чему-то на подобии curl: если передать в него ссылку, то содержимое web-страницы будет скачано и сохранено в файле на локальном диске. К тому же, с помощью воображаемого флага **-a** можно увидеть ход работы.

Команда для выполнения сценария в bash будет выглядеть следующим образом:

getpage.sh -a http://ozi-blog.ru/

Аргументы командной строки инициализируются в том же порядке: **$0 = getpage.sh**, **$1 = -a**, **$2 = http://ozi-blog.ru** и их общее количество **$# = 2**.

Стоит помнить, что **$#** – количество всех аргументов, а не сумма слов в команде. Можно подумать, что если скрипт вызывается без каких-либо параметров, то **$#** должно равняться 1 (учитывая имя сценария), однако на самом деле **$#=0**.

Подобная особенность нумерации существует с зари развития UNIX и известна под названием “проблема нулевого индекса”. Так, в массивах первое значение можно получить по индексу 0 или array[0]. Для многих разработчиков это понятно, для других же – вызывает затруднение. Начинающие программисты могут даже игнорировать нулевой индекс в языке C и начинать использовать ячейки массива с 1, а не 0.

После успешной проверки на наличие флага **-a** для первого аргумента ($1) следует переместить все значения остальных параметров ($2) на одну позицию влево (**$1=$2**). Тогда в остальной части программы можно будет считать, что в $1 находится адрес ссылки. Отпадает необходимость избыточных проверок на наличие или отсутствие флага.

Это можно сделать с помощью команды сдвига (**shift**). В итоге получаем **правильный способ обработки одного опционального аргумента** командной строки в shell-скрипте:

flag\_a=0 if [ "$1" = "-a" ]; then flag\_a=1 shift fi url=$1

Однако, иногда возникает ситуация, когда за опциональным аргументом следует обязательный. К примеру, дополним наш воображаемый скрипт флагом **-o**, после которого необходимо указать путь к файлу для сохранения страницы.

Решить эту задачу можно немного изменив предыдущий пример:

outputspecified=0 if [ "$1" = "-o" ]; then outputspecified=1 outputfilename="$2" shift 2 fi

Таким образом, команда shift принимает одно число. Оно указывает на сколько позиций необходимо сдвинуть значения аргументов командной строки. Это легко продемонстрировать, если вывести их до и после выполнения блока кода с **shift 2**:

$ sh getpage.sh -o test.html какая-то-ссылка $# = 3 $1 = -o $2 = test.html $3 = какая-то-ссылка ----- выполнение команды shift 2 ----- $# = 1 $1 = какая-то ссылка $2 = $3 =

Изначально все три аргумента заданы и их количество (**$#**) равно 3. После выполнения **shift 2** все значения сместились на две позиции и их количество также уменьшилось.

Дальнейшее увеличение количества входных параметров значительно усложнит код, потребуется другой подход.

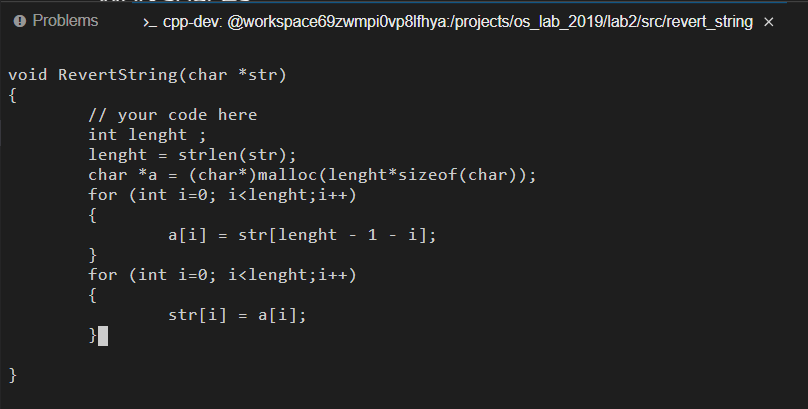
## Обработка большого количества аргументов командной строки в Bash

Даже обработка двух флагов сопряжена с трудностями. Качества кода выше значительно ухудшится, если аргументы могут указываться в произвольном порядке. Но даже без этого, для обработки трёх стартовых условий (**-a**, **-c**, **-o**) перед операцией сдвига понадобится проверка всех вариантов. Потом еще раз проверка и сдвиг. И еще раз проверка и сдвиг. Кроме того, пользователи любят комбинировать опции запуска программы между собой. Представьте, как ужасно разбирать на составные части входные данные вида **-ac -o** или **-aco**.

Задание

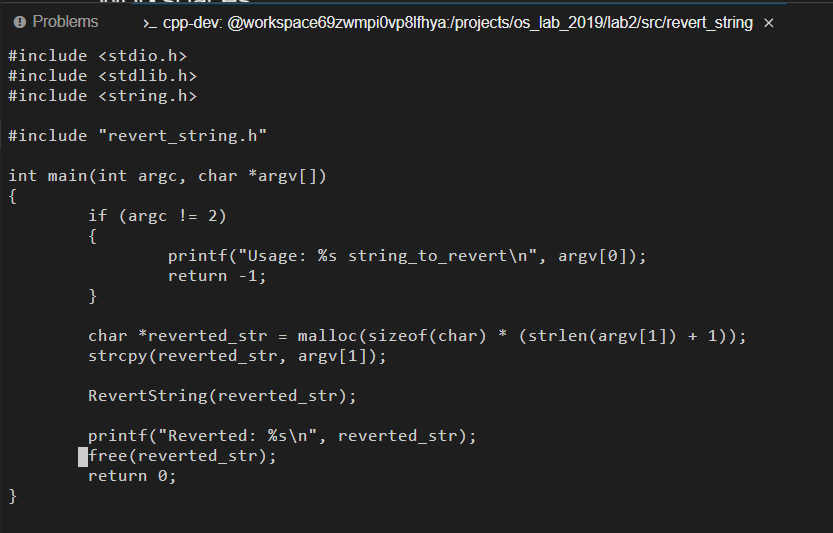
Revert\_string.c

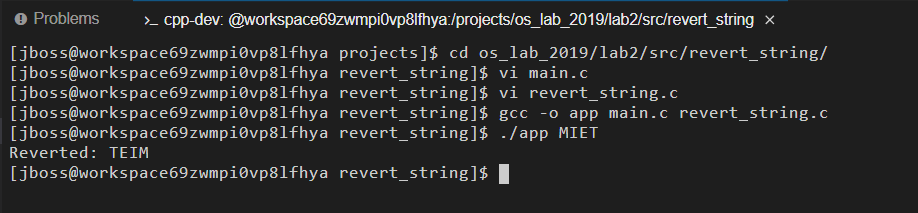
создает переменную **lenght** для хранения длины строки и временную переменную (перевернутая строка). Выделяет динамическую память переменной **a**. Копирует обратный элемент строки str в строку **a**. Сохранить строку **str = а**,то мы перевернули строку



Main.c

После передачи аргумента программе мы выделим динамическую память для переменной reverted\_str и скопируем аргумент в переменную. Функция RevertString перевернет строку reverted\_str. Экспорт reverted\_str у нас есть перевернутая строка



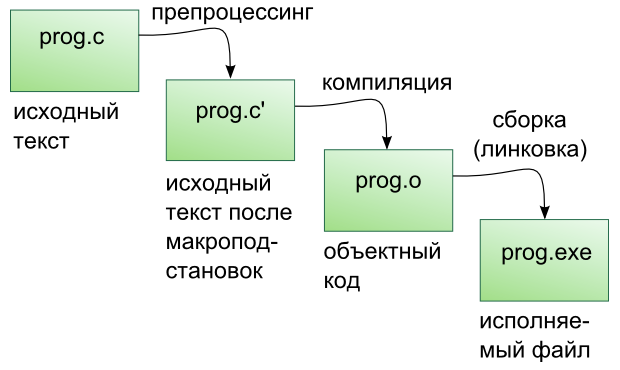


## **Задание 3:**

### Необходимые знания

# Основы работы компилятора: препроцессор, компилятор, линковщик. Их роли и порядок работы.

Компиляция исходных текстов на Си в исполняемый файл происходит в три этапа.



**Препроцессинг**

Эту операцию осуществляет **текстовый препроцессор**.

Исходный текст частично обрабатывается — производятся:

* Замена комментариев пустыми строками
* Текстовое включение файлов — #include
* Макроподстановки — #define
* Обработка директив условной компиляции — #if, #ifdef, #elif, #else, #endif

**Компиляция**

Процесс компиляции состоит из следующих этапов:

1. **Лексический анализ.** Последовательность символов исходного файла преобразуется в последовательность лексем.
2. **Синтаксический анализ.** Последовательность лексем преобразуется в дерево разбора.
3. **Семантический анализ.**Дерево разбора обрабатывается с целью установления его семантики (смысла) — например, привязка идентификаторов к их декларациям, типам, проверка совместимости, определение типов выражений и т. д.
4. **Оптимизация.**Выполняется удаление излишних конструкций и упрощение кода с сохранением его смысла.
5. **Генерация кода.** Из промежуточного представления порождается объектный код.

Результатом компиляции является **объектный код.**

**Объектный код** — это программа на языке машинных кодов с частичным сохранением символьной информации, необходимой в процессе сборки.

При **отладочной сборке** возможно сохранение большого количества символьной информации (идентификаторов переменных, функций, а также типов).

**Компоновка**

Также называется **связывание**, **сборка** или **линковка**.

Это последний этап процесса получения исполняемого файла, состоящий из **связывания воедино всех объектных файлов проекта**.

При этом возможны ошибки связывания.

Если, допустим, функция была объявлена, но не определена, ошибка обнаружится только на этом этапе

# Что такое статическая и динамическая линковка. В чем разница?

* **Статическая библиотека** (или ***«архив»***) состоит из подпрограмм, которые непосредственно компилируются и линкуются с вашей программой. При компиляции программы, которая использует статическую библиотеку, весь функционал статической библиотеки (тот, что использует ваша программа) становится частью вашего исполняемого файла. В Windows статические библиотеки имеют расширение **.lib**(сокр. от *«****lib****rary»*), тогда как в Linux статические библиотеки имеют расширение **.a** (сокр. от *«****a****rchive»*).
* Одним из преимуществ статических библиотек является то, что вам нужно распространить всего лишь 1 (исполняемый) файл, чтобы пользователи могли запустить и использовать вашу программу. Поскольку статические библиотеки становятся частью вашей программы, то вы можете использовать их подобно функционалу своей собственной программы. С другой стороны, поскольку копия библиотеки становится частью каждого вашего исполняемого файла, то это может привести к увеличению размера файла. Также, если вам нужно будет обновить статическую библиотеку, вам придется перекомпилировать каждый исполняемый файл, который её использует.
* **Динамическая библиотека** (или ***«общая библиотека»***) состоит из подпрограмм, которые подгружаются в вашу программу во время её выполнения. При компиляции программы, которая использует динамическую библиотеку, эта библиотека не становится частью вашего исполняемого файла — она ​​так и остается отдельным модулем. В Windows динамические библиотеки имеют расширение **.dll** (сокр. от *«****d****ynamic****l****ink****l****ibrary»* = *«библиотека динамической компоновки»*), тогда как в Linux динамические библиотеки имеют расширение **.so** (сокр. от *«****s****hared****o****bject»* = *«общий объект»*). Одним из преимуществ динамических библиотек является то, что разные программы могут совместно использовать одну копию динамической библиотеки, что значительно экономит используемое пространство. Еще одним преимуществом динамической библиотеки является то, что её можно обновить ​​до более новой версии без необходимости перекомпиляции всех исполняемых файлов, которые её используют.
* Поскольку динамические библиотеки не линкуются непосредственно с вашей программой, то ваши программы, использующие динамические библиотеки, должны явно подключать и взаимодействовать с динамической библиотекой. Этот механизм не всегда может быть понятен для новичков, что может затруднить взаимодействие с динамической библиотекой.

Статическая библиотека это фактически архив объектных файлов, который используется в процессе статической линковки. В результате статической линковки из многих объектных файлов получается один исполняемый, запускается статическая линковка в момент создания исполняемого файла.  
  
Динамическая библиотека это фактически исполняемый файл (т.е. DLL и EXE в Windows имеют одинаковый формат). Динамическая линковка запускается в момент создания процесса (когда вы запускаете исполняемый файл на выполнение), линкуются между собой несколько исполняемых файлов каждый раз, когда создается новый процесс. Так же возможна динамическая линковка уже после запуска, т.е. новая библиотека может быть подгружена в адресное пространство ужа работающего процесса.  
  
Сделать из статической библиотеки динамическую в принципе можно - необходимо ее (статически) слинковать в динамическую библиотеку, при этом будет создана динамическая библиотека и статическая стаб-библиотека, которую можно использовать в проекте вместо статической библиотеки, чтобы вызывать функции из динамической библиотеки.

# Опции компилятора: `-I, -L, -l, -shared, -o, -с, -fPIC`

**-I**

Используется для добавления ваших собственных каталогов для поиска заголовочных файлов в процессе сборки

**-L**

Передается компоновщику. Используется для добавления ваших собственных каталогов для поиска библиотек в процессе сборки.

**-l**

Передается компоновщику. Используется для добавления ваших собственных библиотек для поиска в процессе сборки.

**-o файл**

Поместить вывод в файл 'файл'. Эта опция применяется вне зависимости от вида порождаемого файла, есть ли это выполнимый файл, объектный файл, ассемблерный файл или препроцессированный C код.

Поскольку указывается только один выходной файл, нет смысла использовать '-o' при компиляции более чем одного входного файла, если вы не порождаете на выходе выполнимый файл.

Если '-o' не указано, по умолчанию выполнимый файл помещается в 'a.out', объектный файл для 'исходный.суффикс' - в 'исходный.o', его ассемблерный код в 'исходный.s' и все препроцессированные C файлы - в стандартный вывод.

**-fPIC**

Если поддерживается для целевой машины, порождает позиционно независимый код, подходящий для динамической линковки и не имеющий никаких ограничений на размер глобальной таблицы смещений. Эта опция дает отличие на m68k, m88k и Sparc.

Позиционно-независимый код требует специальной поддержки, и, следовательно, работает толька на некоторых машинах.

**-shared**

Производит разделяемый объект, который может затем быть слинкован с другими объектами, чтобы сформироваться исполнимый файл. Только некоторые системы поддерживают эту опцию.

**-c**

Компилировать или ассемблировать исходные файлы, но не линковать. Стадия ликовки просто не выполняется. Конечный вывод происходит в форме объектного файла для каждого исходного файла.

По умолчанию, имя объектного файла делается из имени исходного файла заменой суффикса '.c', '.i', '.s', и.т.д. на '.o'.

Нераспознанные входные файлы, не требующие компиляции или ассемблирования, игнорируются.

# Утилита ar

ar - это утилита для создания документов и управления ими, в основном используется для библиотек двоичных файловых объектов. ar означает Archiver, который можно использовать для создания архивов любого типа для любых целей, но в значительной степени был заменен на tar, и в настоящее время он используется только для создания и обновления файла статической библиотеки.

-d: удалить модуль из архива.

-m: переместить участника в репозиторий.

-p: Распечатать членство в соответствии с архивом.

-q: быстрое подключение.

-r: -Вставить компонент файла для хранения.

-s: Добавить индекс для сохранения.

-a: добавить новый файл к существующему члену репозитория.

Создайте архив с помощью инструмента «ar» со статической библиотекой «libmath.a» с такими файлами «substraction» и «division».

# ar cr libmath.a substraction.o division.o

Чтобы распаковать архив 'ar'

# ar x libmath.a

cpio представляет собой копирование и выход. Cpio - это общий файловый архив для Linux. Он активно используется RedHat Package Manager (RPM) и initramfs ядра Linux, а также является важным инструментом хранения при установке Apple Computer (pax).

-0: читает список имен файлов, заканчивающихся нулевым символом, вместо новой строки.

-a: сбросить время доступа.

-A: Подключиться.

-B: поменять местами.

-d: создать каталог.

Создайте архивный файл cpio.

# cd tecmint

# ls file1.o file2.o file3.o

# ls | cpio -ov > /path/to/output\_folder/obj.cpio

Чтобы извлечь файл архива cpio

# cpio -idv < /path/to folder/obj.cpio

Переменная окружения `LD\_LIBRARY\_PATH`

LD\_LIBRARY\_PATH - это предопределенная переменная окружения в Linux/Unix, которая задает путь, на который должен ссылаться компоновщик, при связывании динамических библиотек/разделяемых библиотек.

LD\_LIBRARY\_PATH содержит список путей, разделенных двоеточиями, и компоновщик дает приоритет этим путям по стандартным путям библиотеки /lib и /usr/lib. Стандартные пути будут по-прежнему выполняться, но только после исчерпания списка путей в LD\_LIBRARY\_PATH.

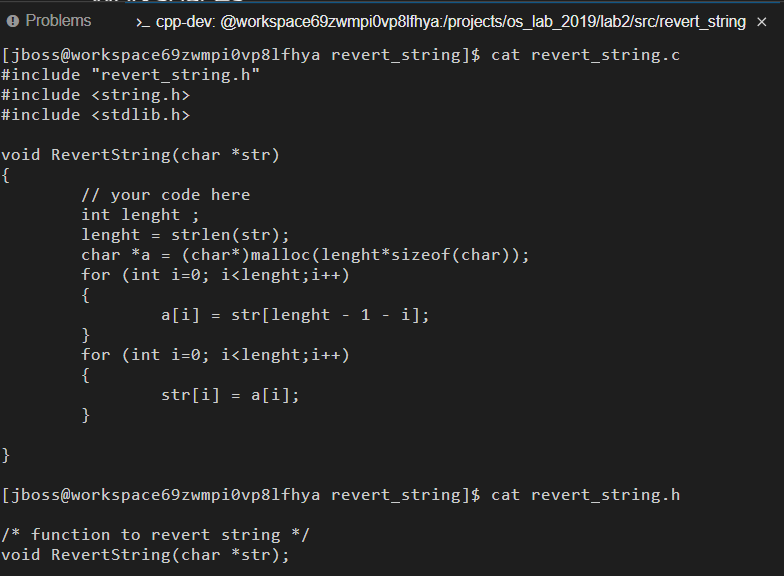
Лучший способ использовать LD\_LIBRARY\_PATH - установить его в командной строке или script непосредственно перед выполнением программы. Таким образом, новый LD\_LIBRARY\_PATH изолирован от остальной части вашей системы.

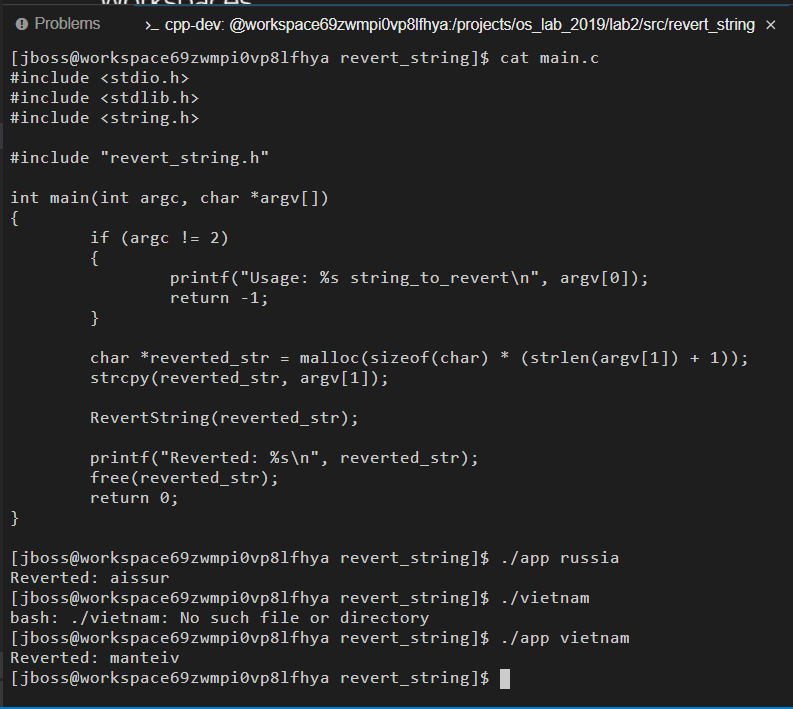
Пример использования:

$ export LD\_LIBRARY\_PATH="/list/of/library/paths:/another/path"

$ ./program

# Задание





Динамическaя  библиотека с RevertString

