Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ»

КАФЕДРА «ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ЭВМ И ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ»

ОТЧЕТ ПО УЧЕБНОЙ ПРАКТИКЕ

Студент **Палладий Евгений Игоревич**

Группа **ИУ7-21Б**

Тип практики **Проектно-технологическая практика**

Название предприятия **НУК ИУ МГТУ им. Н. Э. Баумана**

Студент _____ _____

Руководитель практики _____ **Ломовской И. В.**

Руководитель практики _____ **Кострицкий А. С.**

Оценка _____

2024 г.

Оглавление

ВВЕДЕНИЕ

Цель работы – научиться автоматизировать процессы сборки и тестирования. Изучить процессы получения исполняемого файла. Проработать на практике отладчик gdb для поиска причин ошибок в приложении. Изучить, как на практике представляется многомерный статический массив. Изучить, как в памяти представлена строка языка СИ. Изучить разные способы хранения массива слов в языке Си. Изучить, как располагаются в памяти локальные переменные, структуры и объединения. Выполнить замерный эксперимент

Задание 1.2

Условие

1. Реализовать скрипты отладочной и релизной сборок
2. Реализовать скрипты отладочной сборки с санитайзерами.
3. Реализовать скрипт очистки побочных файлов.
4. Реализовать компаратор для сравнения последовательностей действительных чисел, располагающихся в двух текстовых файлах, с игнорированием остального содержимого
5. Реализовать компаратор для сравнения содержимого двух текстовых файлов, располагающегося после первого вхождения подстроки «Result: ␣».
6. Реализовать скрипт pos_case.sh для проверки позитивного тестового случая по определённым далее правилам.
7. Реализовать скрипт neg_case.sh для проверки негативного тестового случая по определённым далее правилам
8. Обеспечить автоматизацию функционального тестирования

Нетривиальные моменты

Я использовал команды echo -e и sed для форматированного вывода. Пример:

```
echo -e "$(basename "$in_file"): \e[31mFAILED\e[0m"
echo -e "Input data:\n$(sed 's/^/\t/' "$in_file")"
echo -e "Actual output:\n$(sed 's/^/\t/' output.txt)"
```

- Флаг -e позволяет интерпретировать управляющие символы, такие как \n для новой строки и \e[31m для изменения цвета текста.
- \e[31m: Специальная последовательность, которая меняет цвет текста на красный, что полезно для визуального выделения ошибок.
- Команда basename извлекает имя файла из полного пути.
- $(sed 's/^/\t/' "$in_file"): Команда sed используется для добавления табуляции в начале каждой строки файла, что улучшает читаемость вывода.

Выводы

В результате выполнения задания я смог автоматизировать функциональное тестирование однофайлового проекта без аргументов командной строки Автоматизация функционального тестирования с помощью скриптов отладочной и релизной сборок, а также скриптов с санитайзерами, облегчает

процесс проверки работоспособности программы на различных этапах разработки. Реализация позитивных и негативных тестовых случаев позволяет более полно охватить возможные сценарии использования программы. В результате выполнения данного задания я приобрел навыки автоматизации тестирования, что является важным компонентом разработки программного обеспечения.

Задание 1.2

Условие

1. Реализовать скрипты отладочной и релизной сборок
2. Реализовать скрипты отладочной сборки с санитайзерами.
3. Реализовать скрипт очистки побочных файлов.
4. Реализовать компаратор для сравнения последовательностей действительных чисел, располагающихся в двух текстовых файлах, с игнорированием остального содержимого
5. Реализовать скрипт pos_case.sh для проверки позитивного тестового случая по определённым далее правилам.
6. Реализовать скрипт neg_case.sh для проверки негативного тестового случая по определённым далее правилам
7. Обеспечить автоматизацию функционального тестирования.

Выводы

В результате выполнения задания я смог автоматизировать функциональное тестирование многофайлового проекта с аргументами командной строки. В которых передаются как текстовые, так и бинарные файлы. Автоматизация функционального тестирования с помощью скриптов отладочной и релизной сборок, а также скриптов с санитайзерами, облегчает процесс проверки работоспособности программы на различных этапах разработки. Реализация позитивных и негативных тестовых случаев позволяет более полно охватить возможные сценарии использования программы. В результате выполнения данного задания я приобрел навыки автоматизации тестирования, что является важным компонентом разработки программного обеспечения.

Задание 2

Условие

Самостоятельно произвести этапы получения исполняемого файла: препроцессирование, трансляция, ассемблирование, компановка,

Выводы

В ходе выполнения этой лабораторной работы я подробно изучил процесс создания исполняемого файла на языке C.

Используя компиляторы gcc и clang с ключами -v и -save-temps, я проанализировал этапы компиляции. Ключ -v позволил увидеть подробный вывод процесса компиляции, включая команды запуска различных утилит. Ключ -save-temps сохранил промежуточные файлы на каждом этапе, такие как препроцессорный файл, объектный файл и ассемблерный код.

Процесс компиляции был разделён на несколько этапов:

1. Препроцессинг: Обработка директив #include и #define, удаление комментариев и создание расширенного исходного файла.
2. Компиляция: Преобразование расширенного исходного файла в ассемблерный код.
3. Ассемблирование: Преобразование ассемблерного кода в объектный файл.
4. Сборка (линковка): Объединение объектных файлов и библиотек для создания исполняемого файла.

Анализ вывода компилятора позволил мне увидеть параметры,

передаваемые различным утилитам на каждом этапе. Перенаправление вывода в файл упростило анализ и выделение команд. Проверка правильности выделенных команд путём их самостоятельного выполнения подтвердила моё понимание процесса.

Эта лабораторная работа дала мне понимание внутреннего устройства компиляции и сборки программ, а также навыки работы с инструментами командной строки, используемыми в этом процессе.

Задание 3.1

Условие

С помощью отладчика gdb найти ошибки в приложенных программах. Описать процесс отладки

Выводы

Для выполнения задания по поиску ошибок в приложенных программах с помощью отладчика gdb, я следовал следующему процессу:

1. **Запуск gdb:** Сначала я запускал отладчик gdb, передав в него исполняемый файл программы: gdb -silent ./app.exe. “-silent” ужен для того, чтобы е выводилась информация об отладчике

2. **Установка точек останова:** Я установлива точки останова в ключевых местах программы, таких как начало функции main, проблемные функции: break print_bin_file или на определенных строках: break 10. Также я использовал условные точки останова, например: break 10 if n > 10. Что было достаточно удобным средством

3. **Запуск программы:** Используя команду run, я запускал программу с аргументами, чтобы воспроизвести ошибку.

4. **Пошаговое выполнение:** Я использовал команды step и next для пошагового выполнения кода и отслеживания его выполнения.

5. **Проверка переменных**: С помощью команд print, info locals, info

args я проверял значения переменных

Этот процесс позволил мне успешно выявить и исправить ошибки в программе. Я до сих пор использую отладчик gdb, поэтому это задания было для меня очень полезными

Задание 3.2

Условие

1. Изучить, как в памяти представлен многомерный статический массив, для чего: 1. Опишите трёхмерный массив целых чисел, размеры которого равны 2, 3 и 4 соответственно.
2. Показать дамп памяти, который содержит этот массив полностью.
3. Показать, из каких компонент состоит этот массив, постепенно фиксируя размерность массива. Соответствующие компоненты необходимо показать в дампе.
4. Описать указатели для работы с этими компонентами. Чему равен размер элемента соответствующего компонента?

Выводы

В результате выполнения задания я на практике увидел, что в СИ многомерные массивы хранятся как линейные блоки памяти, что упрощает их обработку, но усложняет поиск ошибок, в случае выхода за границы массива. Дамп памяти помог визуализировать представление многомерных массивов. Я понял, как рассчитывать размер соответствующего компонента многомерного массива: произведение количества элементов на их тип.

Задание 3.3

Условие

Изучить, как в памяти представлена строка языка и массив строк в языке программирования Си. Показать дамп памяти

Выводы

В результате выполнения задания я на практике увидел, как представлены строки и массивы указателей на строке в языке СИ. Строки являются массивами символов, заканчивающимися нулевым символом '\0'. Массивы указателей на строки представляют собой массивы, каждый элемент которых является указателем на первый символ строки. Строки размещаются последовательно в памяти, а массивы указателей на строки содержат адреса строк. Это задание помогло мне понять, как представлены строки в языке СИ

Задание 3.4

Условие

Локальные переменные

1. Описать несколько локальных переменных разных типов.
2. На дампе памяти показать, как они располагаются в памяти.
3. Проанализировать зависимость значения адреса переменной от её размера.

Структуры

1. Описать структуру, содержащую несколько полей разного типа.
2. Показать дамп памяти, который содержит эту структуру. На дампе покажите расположение каждого поля структуры.
3. Проанализировать зависимость значения адреса поля от его размера.
4. По какому адресу располагается переменная структурного типа? Какое поле структуры повлияло на значение этого адреса?
5. Упаковать структуру и выполните предыдущие пункты для упакованной структуры.

Объединения

1. Описать объединение, содержащее несколько полей разного типа.
2. Показать дамп памяти, который содержит это объединение. На дампе покажите поле, которому присвоено значение.
3. Присвоить значение другому полю объединения и выполните предыдущий пункт еще раз.

Выводы

В результате выполнения лабораторной работы я приобрёл ценные знания о размещении локальных переменных, структур и объединений в памяти.

Локальные переменные

Локальные переменные различных типов не обязательно размещаются в памяти последовательно, компилятор имеет права разместить их как захочет. Это необходимо для увеличения оптимизации программы. Адреса памяти переменных зависят от их размера.

Структуры

1. Поля структуры обязательно располагаются в памяти последовательно, в отличие от переменных, но в памяти между полями может произойти выравнивание, в результате чего появятся “пустые” байты. Адреса полей структуры зависят от их размеров и выравнивания. Поля с большими размерами могут влиять на выравнивание последующих полей. Упаковка структуры устраняет выравнивание, что позволяет более компактно размещать поля в памяти, но может замедлить доступ к данным из-за отсутствия выравнивания.

Объединения

Все поля объединения занимают одно и то же место в памяти, причём размер объединения определяется размером самого большого поля. При присвоении значений разным полям объединения, одно и то же место в памяти интерпретируется по-разному, что позволяет эффективно использовать память, но требует осторожности при

работе с данными.

Эти знания помогли мне лучше понимать, как устроены в памяти локальные переменные, структуры и объединения.

Задание 4

Условие

Провести сравнение производительности работы программы для разных способов работы с элементами одномерного массива:

- использование операции индексации a[i];
- формальная замена операции индексации на выражение *(a + i);
- использование указателей для работы с массивом.

Измерение производительности необходимо выполнить двумя способами:

1. Реализовать инфраструктуру измерения времени выполнения функции в самой программе.
2. Реализовать инфраструктуру измерения времени выполнения функции вне программы. В этом случае внутри программы выполняется замер одного выполнения функции (время выполнения функции выводится на экран или в файл), а повторные замеры выполняются путем многократного запуска самой программы

Нетривиальные моменты

В скрипте update_data я сравнивал значение rse, но так как оно вещественное, а в bash сравнение вещественных чисел не поддерживается напрямую встроенными операторами сравнения. Поэтому я использовал команду bc (Basic Calculator), которая поддерживает арифметические операции с плавающими точками.

```
if (( $(echo "$rse_value < 1.0" | bc -l) )); then
  break
fi
```

Выполнение bc:

- | bc -l передаёт это выражение в bc с флагом -l, который включает стандартную математическую библиотеку.
- bc вычисляет выражение и возвращает 1 (истина), если условие верно, или 0 (ложь), если условие неверно.

Выводы

После проведения замеров, сбора и обработки данных о времени выполнения программ я сделал выводы, что конкретно на моей машине при замерах внутри использование указателей менее эффективно по времени, чем использование индексация или замены индексации. Индексация и замена индексации имеют почти одинаковую эффективность времени выполнения при разных размерах массивов. А при замерах вне программы, использование указателей лишь чуть-чуть хуже по времени, чем использование индексации или замены индексации. Индексация и замена индексации имеют практически одинаковую эффективность. Время выполнения сортировки при замерах вне программы немного больше, чем при замерах внутри программы

ЗАКЛЮЧЕНИЕ

В ходе выполнения задач по проектно-технологической практике я научился: автоматизировать процессы сборки и тестирования. Изучил процессы получения исполняемого файла. Научился использовать отладчик gdb для поиска причин ошибок в приложениях. Изучил, как на практике представляется многомерный статический массив. Изучил, как в памяти представлена строка языка СИ. Изучил разные способы хранения массива слов в языке Си. Изучил, как располагаются в памяти локальные переменные, структуры и объединения. Выполнил замерный эксперимент