Федеральное агентство связи

Сибирский Государственный Университет

Телекоммуникаций и Информатики

**Кафедра вычислительных систем**

**КУРСОВАЯ РАБОТА №1**

**по дисциплине**

**«Технология разработки программного обеспечения»**

**Выполнили: Зиновьев М. И.**

**Лантушенко М.В.**

**Группа: ЗП-91**

**Номер студенческого билета:**

**73190120**

**73190086**

**Адрес электронной почты:**

[**M.Zinov33Ev@yandex**](mailto:M.Zinov33Ev@yandex)**.**

**lantushenko2013@gmail.com**

**Проверил: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Новосибирск, 2020 г.**

В репозитории рассматривается настройка базового пакета тестирования со значками github для библиотеки C с использованием Travis-CI.

**Мотивация**

Я просто хотел создать небольшой автономный тестовый проект, чтобы увидеть инструменты и рабочий процесс для тестирования языка C (или C ++).

**Характеристики**

Файлы lib.h / lib.c разбиты на примеры тестирования встроенной библиотеки.

**Как это работает**

В этом демонстрационном проекте есть библиотека C. Код библиотеки - это всего лишь несколько демонстрационных функций, которые находятся в файлах lib.h и lib.c. На самом деле они ничего не делают, но позволяют просто абстрагироваться от того, что необходимо для создания более крупного проекта.

**Модульное тестирование**

Модульное тестирование - это практика написания небольших тестов, чтобы увидеть, как этот фрагмент кода, обычно полный модуль или библиотека, проходит набор тестов, чтобы убедиться, что он работает должным образом. Простые модульные тесты выполняются после написания функции. Затем мы создаем небольшую программу модульного тестирования, которая вызывает нашу новую функцию с таким количеством различных параметров, которое мы считаем подходящим, чтобы убедиться, что она работает правильно. Если возвращенные результаты соответствуют ожидаемым результатам, мы можем сказать, что функция прошла проверку. Если результаты для данного набора параметров не совпадают, мы вызываем, которое записывает сбой и пытается продолжить выполнение, а затем тестирует в нашей тестовой программе. Цель состоит в том, чтобы создать набор этих тестов, которые проверяют все возможные пути выполнения в нашем коде и проходят все тесты.

Например, если есть такая функция:

int add5ifGreaterThan2 ( int a) {

int r;

если (а> 2 )

г = а + 5 ;

еще

г = а;

return r;

}

Наша тестовая программа должна предоставлять значения a, которые меньше и больше 2, поэтому оба пути оператора if

если (а < 2 )

проверены.

Мы делаем это с помощью тестового кода, такого как:

// код в тестовой программе ...

ASSERT (add5ifGreaterThan2 ( 1 ) == 1) // предоставляет значение 'a', которое проверяет случай if (a <2) и проверяет результат

ASSERT (add5ifGreaterThan2 ( 3 ) == 8) // предоставляет значение 'a', которое проверяет случай if (a> 2) и проверяет результат

Макрос ASSERT проверяет, является ли результат логически верным. Если это не так, запускается процесс ошибки. В зависимости от используемой среды тестирования в случае сбоя оператора могут быть проверены различные типы журналирования и вывода.

**Каркасы**

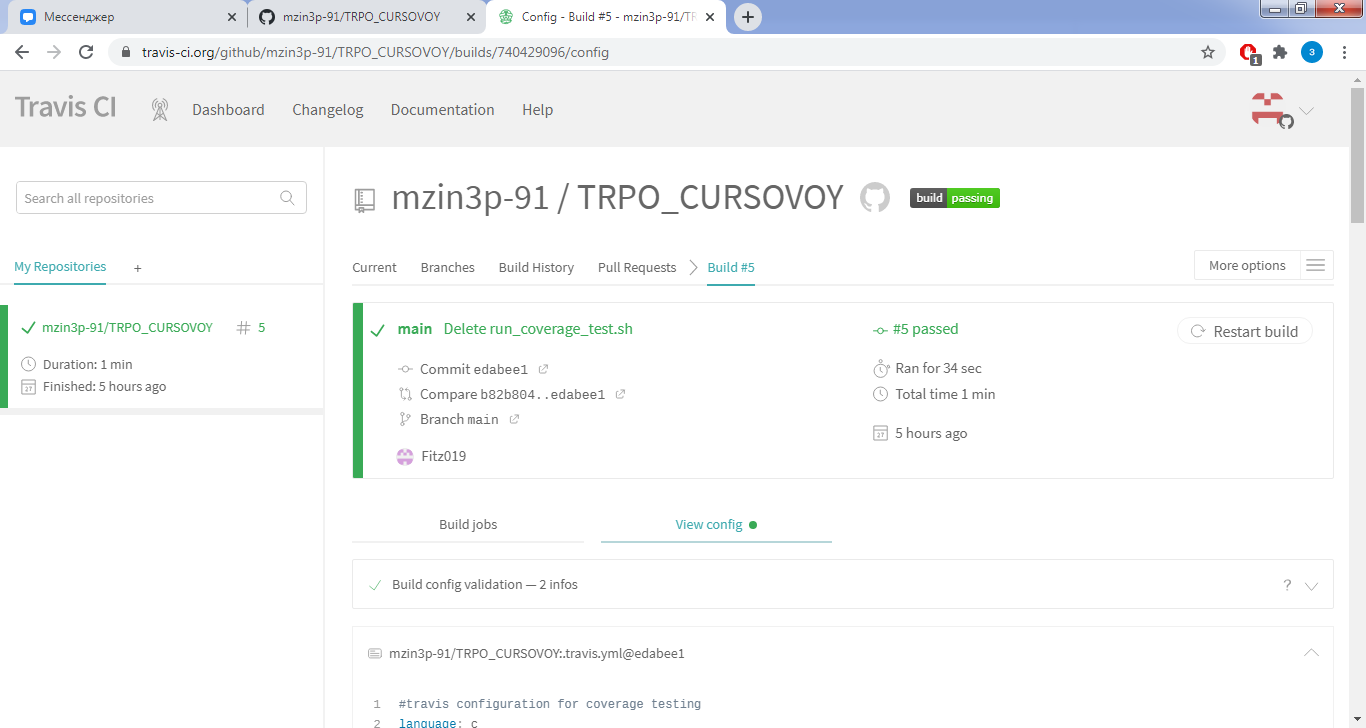
Чтобы упростить автоматизацию модульного тестирования, были написаны фреймворки модульного тестирования, помогающие получать результаты тестирования из вызовов функций, собирать статистику о прохождении / неудаче тестовых случаев.

**Тестирование против непрерывной интеграции**

Файлы lib.h / lib.c разбиты на примеры тестирования встроенной библиотеки.

После того, как вы выполнили модульные тесты и получили код для запуска с использованием локального набора тестов, начинается следующий шаг. Здесь начинается непрерывная интеграция Travis-CI. Когда программа CI запускает набор тестов, ее можно настроить на принятие или отклонение вашего кода в зависимости от прохождения тестов. Это, в свою очередь, можно использовать для автоматического развертывания вашего кода.

Travis-CI просматривает .travis.yml, чтобы узнать, как запустить код. В этом случае он сначала вызывает make, который компилирует lib.c и example.c в lib.o и example.o, а затем связывает их для создания финального исполняемого файла с именем example.out. Функция main () вызывает локальную функцию run\_tests (), которая, в свою очередь, вызывает каждый отдельный тестовый пример. Если run\_tests () сможет успешно запустить все тесты, он вернется в main () со значением S\_OK, иначе он вернет код ошибки. Это значение затем возвращается из программы main () в example.out при выходе.

Затем Travis-CI запускает example.out и ищет код выхода из функции main (). Поскольку это система в стиле Posix, нулевой код выхода из example.out считается проходящим, и, следовательно, Travis-ci объявляет прохождение сборки. Если возвращается ненулевое значение, travis объявит сборку неудачной. Подводя итог, можно сказать, что основным средством для Travis'а узнать, прошел ли набор тестов, является получение правильного кода выхода из исполняемого файла набора тестов, который в нашем случае выполняет example.out. 

**Покрытие кода**

Покрытие кода достигается с помощью gcov из набора тестов gcc. Тестовая программа example.out скомпилирована с флагами -ftest-охват -fprofile-arcs. Чтобы увидеть покрытие кода, запустите gcov:

очистить

сделать

./test-library.out

gcov lib.c

который сгенерирует файл

lib.c.gcov