гайд по check и его запуску @[**dmagic**](https://app.slack.com/team/U02JZR87X0W)

Открываем терминал и сразу копируем вставляем

1)curl -fsSL <https://rawgit.com/kube/42homebrew/master/install.sh> | zsh

Закрываем терминал и открываем его ещё раз после установки

2)brew install cppcheck

3)brew install check

Добавляем в ваш main.c include <check.h> (VS code будет ругаться, но вы ему не верьте)

(Как компилировать) gcc -c main.c -I../../.brew/include -L../../.brew/Cellar/check/0.15.2/lib -lcheckЕсли у вас уже стоит homebrew, то .brew меняем на homebrew(Но лучше удалить хомбрю и поставить по гайду)

А тут можно взять код для проверки <https://stackoverflow.com/questions/14176180/using-c-unit-testing-framework-check-without-autotools/54095032>

Дополнительно:

@**camaerie** Для того чтобы VS code не ругался и тоже мог найти check.h, можем навести на строку #include с ошибкой -> Quick fix... -> Edit "Include path" setting, в открывшемся окне настроек дописать путь к папке, куда brew кладет заголовочные файлы библиотек [~/.brew/include/\*\*]

Гайд по написанию тестов через Check @[**camaerie**](https://app.slack.com/team/U02K3J260BX)

КАК ИСПОЛЬЗОВАТЬ CHECK

\

Тест как таковой может быть завален или нет. Соответственно во втором случае всё хорошо, в первом с тести**руемой** функцией что-то не такОсновные понятия:

-test function(тест) - тестовая функция, которая заваливается или нет и содержит в себе непосредственно код теста

-test case(кейс) - набор тестовых функций, в нашем случае рекомендую в один кейс класть все тесты

-test suite(сьюит) - аналогично набор кейсов, в нашем случае рекомендую игнорировать и сделать один сьюит на весь проектПринцип: Тестовый файл вполне себе содержит функцию main и #include для тестируемой библиотеки и check.h

Над мейном описываются тесты, а в мейне формируются сьюит и кейсы. К кейсам привязываются тесты, сами кейсы к сьюиту, потом это дело стартует.

#include <check.h>

#include <stdio.h>

#include <signal.h>

#include "s21\_string.h"

START\_TEST(getPassTest) //тест функции get

{

ck\_assert\_msg(get(2) == 2, "failed on 2");

ck\_assert\_msg(get(3) == 3, "failed on 3");

ck\_assert\_msg(get(4) == 4, "failed on 4");

}

END\_TEST

START\_TEST(getFailTest) //Конечно, на практике мы не пишем тесты, обреченные на провал

{

ck\_assert\_msg(get(2) == 12, "failed on 12");

ck\_assert\_msg(get(3) == 321, "failed on 321");

ck\_assert\_msg(get(4) == 222, "failed on 222");

}

END\_TEST

START\_TEST(busError) //Тест, возвращающий bus error

{

int a = busE(); //Гарантированно вызовет bus error

}

END\_TEST

int main(void)

{

Suite \*s1 = suite\_create("Core"); //Объявляем сьюит

SRunner \*sr = srunner\_create(s1); //Лаунчер сьюита

int nf; // Статус тестов сьюита

TCase \*getCase = tcase\_create("sumBaseTest"); //Новый кейс getCase для функции get

suite\_add\_tcase(s1, getCase); //Добавляем кейс getCase для запуска

//В кейс getCase добавляем тесты, описанные выше

tcase\_add\_test(getCase, getPassTest);

tcase\_add\_test(getCase, getFailTest);

TCase \*busECase = tcase\_create("busErrorTest"); //Новый кейс busECase для другой функции

//Задаем, что тест busError хранится в busECase и должен выдать bus error

tcase\_add\_test\_raise\_signal(busECase, busError, SIGBUS); //SIGBUS - код ошибки bus error из <signal.h>

tcase\_add\_test\_raise\_signal(busECase, busError, 11); //В дополнение тот же тест с другим ожидаемым кодом для примера ошибки

suite\_add\_tcase(s1, busECase); //Можно добавлять кейс и после привязки к нему тестов

//Запустить всё это дело

srunner\_run\_all(sr, CK\_ENV);

nf = srunner\_ntests\_failed(sr);

srunner\_free(sr);

return nf == 0 ? 0 : 1;

О командах:

ck\_assert - функции для теста всякого, ck\_assert\_msg валит тест если выражение первым аргументом вернуло false и выводит в этом случае текст второго аргумента

Полный список - https://libcheck.github.io/check/doc/check\_html/check\_4.html#Convenience-Test-Functions

START\_TEST/END\_TEST - для написания тестов, они не пишутся просто как функции

tcase\_add\_test - добавить тест к кейсу

tcase\_add\_test\_raise\_signal - добавить тест к кейсу, но ожидается что тест вернет нужный сигнал ошибки, иначе завалится. это способ проверять, что функция выдает segmentation fault или ещё что в таком духе, за списком кодов в <signal.h>

гайд о том, как настроить VSCode чтобы она форматировала код под линтер @[**dtammie**](https://app.slack.com/team/U02K6HBKQBV) / @[**Aglendon**](https://app.slack.com/team/U02JRSQ9F55)

Скачиваем расширение С/С++

1. Настройки - Tab Size - ставим 4
2. Там же в настройках в поиске вбиваем “с\_срр” в 2х пунктах выбираем стиль Google
3. Там же в настройках в поиске ищем “format on save” ставим галку

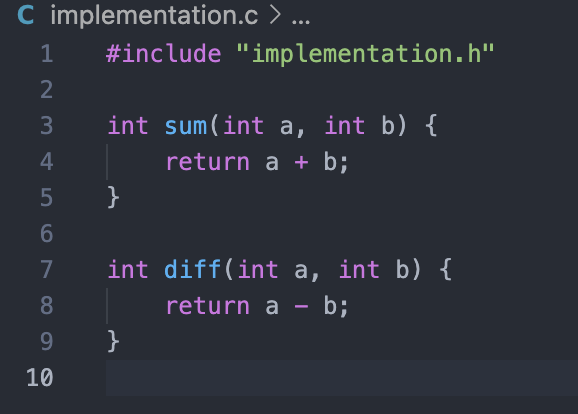
корректная строка для “C\_Cpp: Clang\_format\_fallback Style” и “C\_Cpp: Clang\_format\_style” - {BasedOnStyle: Google, UseTab: Never, IndentWidth: 4, TabWidth: 4}

**Всем тем, кто до сих пор пишет тесты вручную посвящается. @**phawkgir

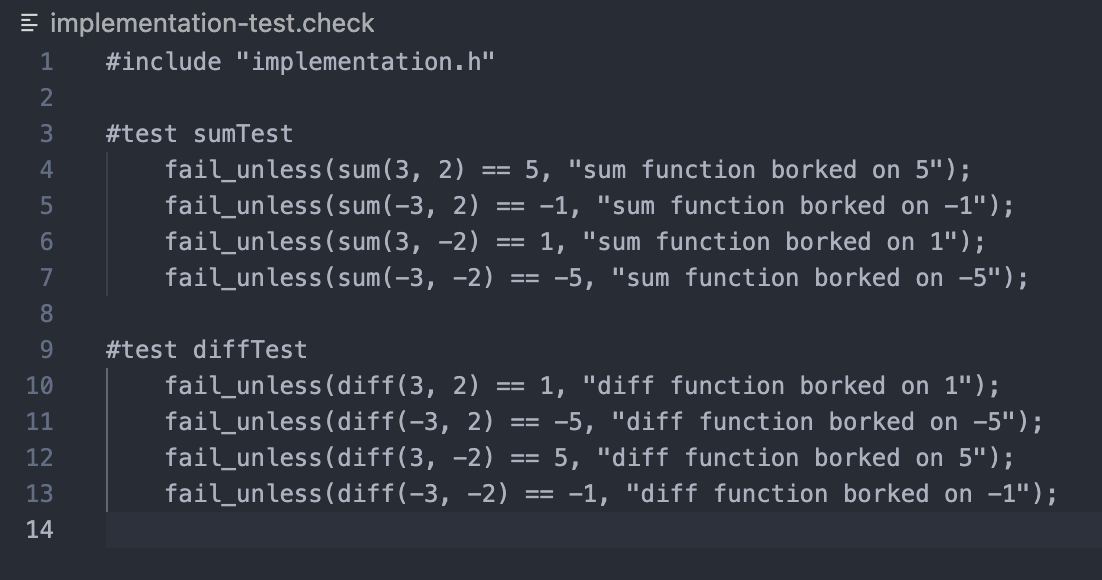
С библиотекой check поставляется также супер-крутая удобная утилита checkmk, которая лежит в папке установки check в bin/ и которая позволяет вам избавиться от рутинного прописывания однообразных строчек в тестовых файлах.

Краткий гайд, как ей пользоваться опишу в треде.

Допустим, у меня есть реализация простейших функций суммирования и вычитания sum() и diff(). Заголовки этих функций вынесены в implementation.h



Я могу создать файл-макрос подобного вида с описанием всех тестов, которые я хочу преобразовать в тестовый сишный файл. Он будет иметь следующий вид:



Далее все, что мне нужно - это запустить утилиту checkmk и передать ей этот файл. В обычных условиях, исходные файлы библиотеки check устанавливаются в директорию /bin, поэтому запуск команды мог бы выглядеть следующим образом:  
checkmk implementation-test.check > implementation-test.c  
Но так как мы устанавливали check через homebrew, то нужно прописать полный путь к исполняемому файлу в таком виде:  
~/homebrew/Cellar/check/0.15.2/bin/checkmk implementation-test.check > implementation-test.c  
На выходе получаем готовый файл для тестов implementation-test.c  со следующим содержимым:



Выложу краткую памятку по дебагу в VS Code @rdexter

Установка

1. Создаём папки .vscode и build в корне проекта (если их нет)
2. Копируем в папку .vscode файлы launch.json и tasks.json
3. Создаём Makefile в корне проекта и прописываем флаг -g при компоновке (например gcc -g src/main.c -o build/main.out )

Процесс отладки

1. Ставим брейкпоинт в отлаживаемом файле (тыкаем мышкой правее номера строки)
2. Нажимаем F5
3. Для перехода построчно нажимаем Fn + F11 (чтобы отключить данную комбинацию клавиш с целью сворачивания окон: Заходим в Системные настройки -> Клавиатура -> Вкладка ‘Сочетания клавиш’ -> Убираем галочку с ‘Показать рабочий стол’)
4. Для прохождения по функции не входя в неё Fn + F10
5. Окончание дебага F5
6. Слева будет окно с переменными, удобно отслеживать значения построчно
7. Если не хватает выражений - можно написать дополнительные выражения в Watch, например (char) num

Прикладываю файлы. Они написаны чисто под меня, кому неудобна структура - можете поиграться в настройках launch.json и tasks.json!!! Большая проблема, что на Маке недоступен вывод терминала в режиме отладки (ну то есть не выводит printf’ы). Лечится это очисткой потока перед каждым выводом, но для меня это напряжно) потом забуду убрать перед сдачей проекта :))  
Если есть идеи, как это лечить - напишите в комментах!

ФАЙЛЫ launch.json tasks.json

Дополнительно: @[**cangelen**](https://app.slack.com/team/U02K3HGJN4V) На ubuntu была ошибка: "unable to determine path to debugger". Заработало после изменения параметра "MIMode": "gdb" в launch.json

Памятка по strerror. @[**ldiomede**](https://app.slack.com/team/U02KW7P49EC)

Как оформить сообщения об ошибках в виде макроса и выбрать нужную операционную систему.

С помощью директивы #if #elif #endif выбираем операционную систему defined(\_\_APPLE\_\_) или defined(\_\_linux\_\_),

затем внутри директивы дефайном подключаем массив,

который оформляем в фигурных

скобках(не забываем ставить слэш в конце строки, если делаете перенос)

#define < название массива> {"Undefined error: 0", "Operation not permitted", ……, }Для Мака 0 - 106 ошибок(их можно вывести в консоли используя оригинальную функцию strerror из библиотеки string).

Для Линукс 0 - 133 их можно найти, выведя в онлайн-компиляторе, также используя функцию strerror. https: // [www.onlinegdb.com/online\_c++\_compiler](http://www.onlinegdb.com/online_c++_compiler)Сам массив в коде используем приравняв его к статическому указателю на массив

static char \*array[] = <название массива> ;

И уже можем его обращаться к нему передавая в него код ошибки errnum.

array[errnum]