Шаблон отчёта по лабораторной работе

Простейший вариант

Дмитрий Сергеевич Кулябов

Содержание

[Цель 1](#_Toc73799477)

[Задание 1](#_Toc73799478)

[Ход работы 1](#_Toc73799479)

[Контрольные вопросы 10](#_Toc73799480)

[Вывод 13](#_Toc73799481)

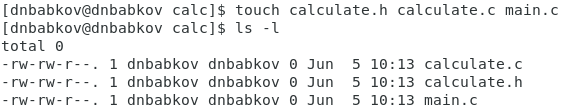
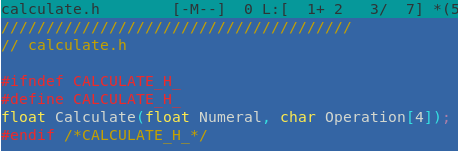
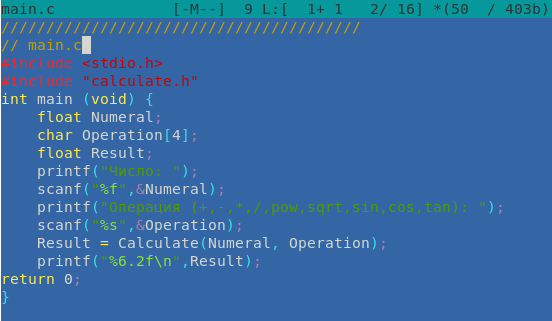
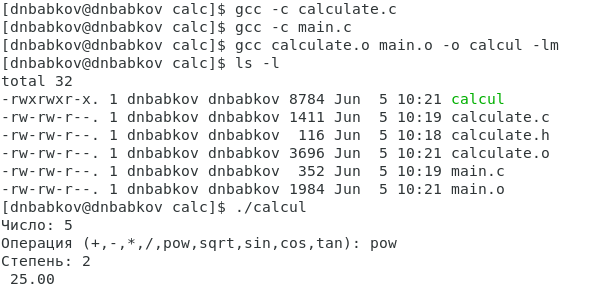
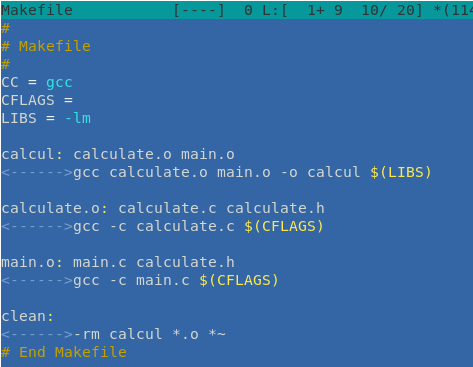
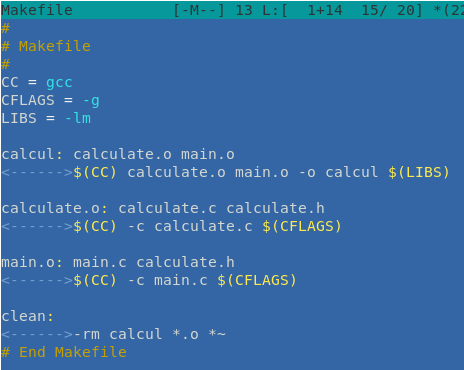
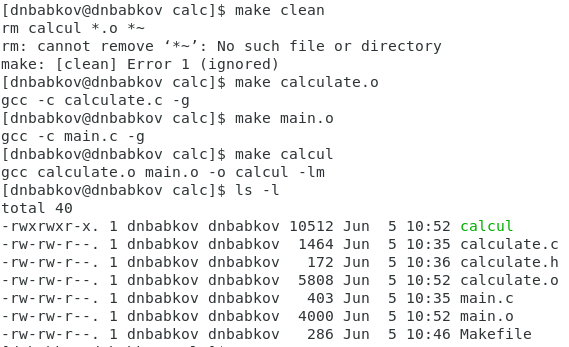
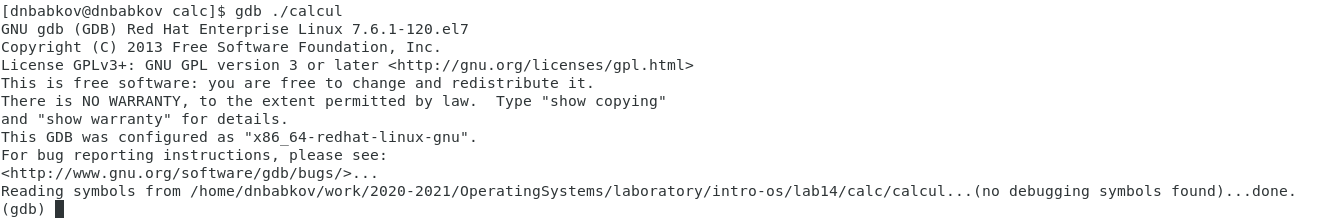
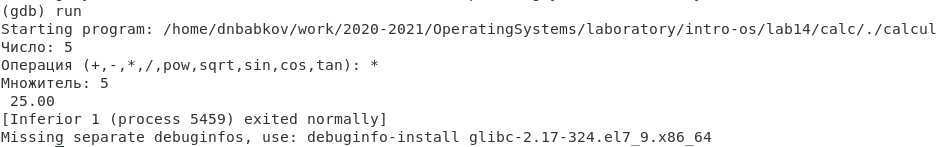
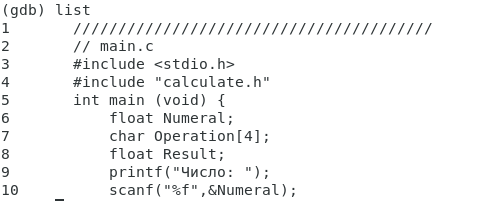
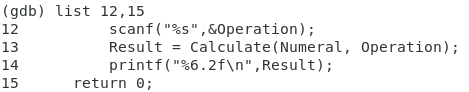
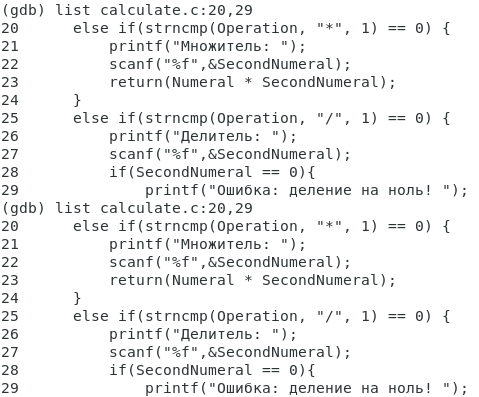
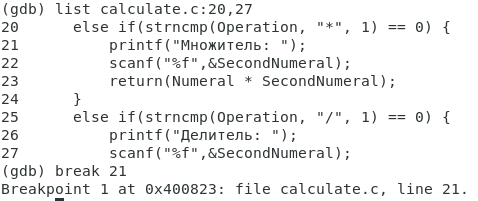
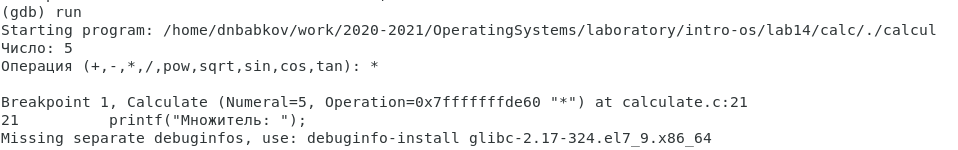
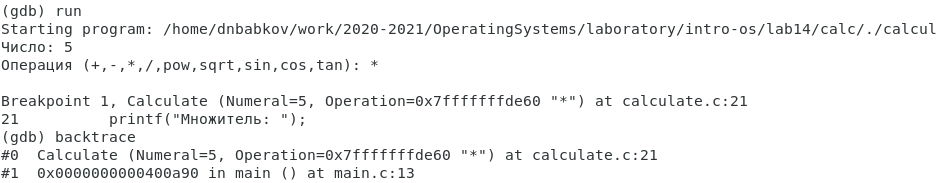
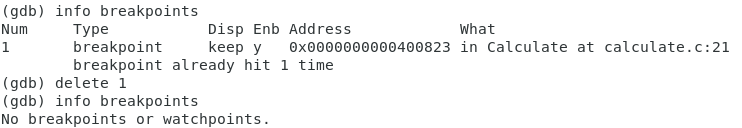
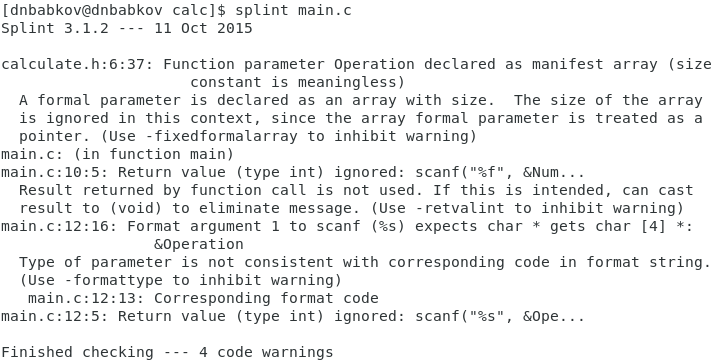
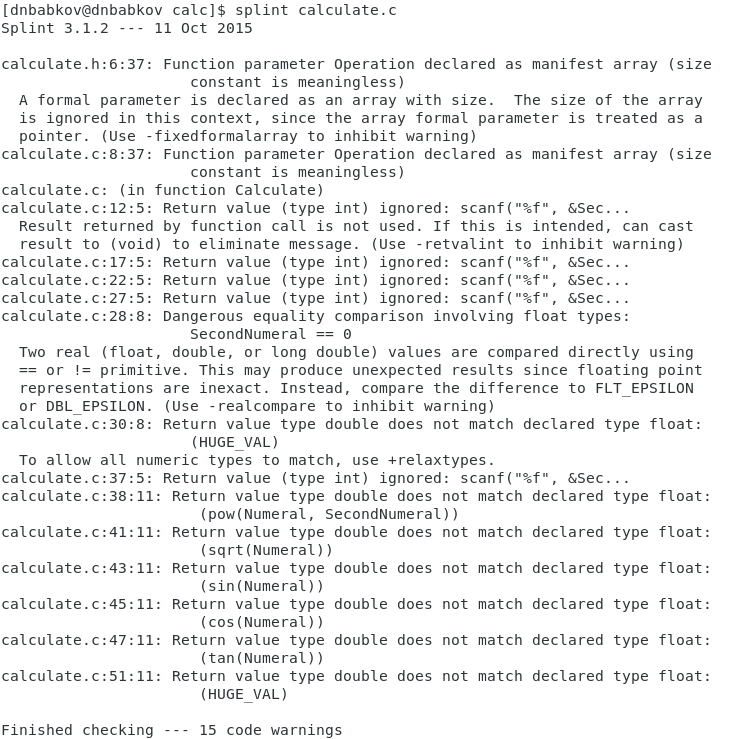
# Цель

Приобрести простейшие навыки разработки, анализа, тестирования и отладки приложений в ОС типа UNIX/Linux на примере создания на языке программирования С калькулятора с простейшими функциями.

# Задание

Написать и отладить программу, выполняющую функции калькулятора

# Ход работы

1. В каталоге **~/work/2020-2021/OperatingSystems/intro-os** создал подкаталог **lab13**, в котором создал файлы **calculate.h**, **calculate.c**, **main.c.** (Рис.1)  
    Рис.1
2. В файлы написал код, данный в методических материалах (Рис.2, 3, 4)  Рис.2  
    Рис.3  
    Рис.4
3. С помощью **gcc** выполнил компиляцию программы (Рис.5)  
    Рис.5
4. Синтаксических ошибок обнаружено не было
5. Создал Makefile с содержанием, данным в методических материалах (Рис.6)  
    Рис.6  
   При вызове **make calcul** Makefile делает из объектных файлов **calculate.o** и **main.o** исполняемый файл **calcul** с помощью команды **gcc calculate.o main.o -o calcul -lm**. При вызове make **calculate.o** из файлов **calculate.c** и **calculate.h** создается объектный файл. То же самое происходит при вызове make **main.o**. При вызове make clear удаляется всё, что создает Makefile.
6. Исправил Makefile (Рис.7)  
    Рис.7  
   После этого сделал с его помощью исполняемый файл **calcul** (Рис.8)  
    Рис.8
   * Вызываю отладчик **gdb** для исполняемого файла с помощью команды **gdb ./calcul** (Рис.9)  
      Рис.9
   * Запускаю программу в среде отладчика с помощью команды **run** (Рис.10)  
      Рис.10
   * С помощью команды **list** вывожу на экран девять первых строк исходного текста программы (Рис.11)  
      Рис.11
   * Вывожу строки с 12 по 15 командой **list12,15** (Рис.12)  
      Рис.12
   * Вывожу строки не основного файла (Рис.13)  
      Рис.13
   * В файле **calculate.c** устанавливаю точку останова на 21 строке с помощью команды **break 21** (Рис.14)  
      Рис.14
   * Вывожу информацию об имеющихся точках останова с помощью команды **info breakpoints** (Рис.15)  
     Рис.15 Рис.15
   * Запускаю программу внутри отладчика, чтобы убедиться, что точка останова работает (Рис.16)  
      Рис.16
   * Вывожу стек вызываемых функций с помощью команды **backtrace** (Рис.17)  
      Рис.17
   * Вывожу значение переменной **Numeral** командой **print Numeral** (Рис.18)  
     Рис.18 Рис.18
   * Вывожу переменную **Numeral** с помощью команды **display Numeral** (Рис.19)  
     Рис.19 Рис.19
   * Удаляю точки останова командой **delete** (Рис.20)  
      Рис.20
7. С помощью утилиты **splint** анализирую файлы **main.c** и **calculate.c** (Рис.21, 22)  
    Рис.22  
    Рис.21

# Контрольные вопросы

1. Чтобы получить информацию о возможностях программ gcc, make, gdb и др. нужно воспользоваться командой man или опцией -help (-h) для каждой команды.
2. Процесс разработки программного обеспечения обычно разделяется на следующие этапы:

* планирование, включающее сбор и анализ требований к функционалу и другим характеристикам разрабатываемого приложения;
* проектирование, включающее в себя разработку базовых алгоритмов и спецификаций, определение языка программирования;
* непосредственная разработка приложения:
  + кодирование − по сути создание исходного текста программы (возможно в нескольких вариантах); – анализ разработанного кода;
  + сборка, компиляция и разработка исполняемого модуля;
  + тестирование и отладка, сохранение произведённых изменений;
* документирование.  
  Для создания исходного текста программы разработчик может воспользоваться любым удобным для него редактором текста: vi, vim, mceditor, emacs, geany и др.  
  После завершения написания исходного кода программы (возможно состоящей из нескольких файлов), необходимо её скомпилировать и получить исполняемый модуль.

1. Для имени входного файла суффикс определяет какая компиляция требуется. Суффиксы указывают на тип объекта. Файлы с расширением (суффиксом) .c воспринимаются gcc как программы на языке С, файлы с расширением .cc или .C − как файлы на языке C++, а файлы c расширением .o считаются объектными. Например, в команде «gcc -c main.c»: gcc по расширению (суффиксу) .c распознает тип файла для компиляции и формирует объектный модуль − файл с расширением .o. Если требуется получить исполняемый файл с определённым именем (например, hello), то требуется воспользоваться опцией -o и в качестве параметра задать имя создаваемого файла: «gcc -o hello main.c».
2. Основное назначение компилятора языка Си в UNIX заключается в компиляции всей программы и получении исполняемого файла/модуля.
3. Для сборки разрабатываемого приложения и собственно компиляции полезно воспользоваться утилитой make. Она позволяет автоматизировать процесс преобразования файлов программы из одной формы в другую, отслеживает взаимосвязи между файлами.
4. Для работы с утилитой make необходимо в корне рабочего каталога с Вашим проектом создать файл с названием makefile или Makefile, в котором будут описаны правила обработки файлов Вашего программного комплекса. В самом простом случае Makefile имеет следующий синтаксис:

<цель\_1> <цель\_2> ... : <зависимость\_1> <зависимость\_2> ...  
<команда 1>  
...  
<команда n>

Сначала задаётся список целей, разделённых пробелами, за которым идёт двоеточие и список зависимостей. Затем в следующих строках указываются команды. Строки с командами обязательно должны начинаться с табуляции. В качестве цели в Makefile может выступать имя файла или название какого-то действия. Зависимость задаёт исходные параметры (условия) для достижения указанной цели. Зависимость также может быть названием какого-то действия. Команды − собственно действия, которые необходимо выполнить для достижения цели. Общий синтаксис Makefile имеет вид:

target1 [target2...]:[:] [dependment1...]  
[(tab)commands] [#commentary]  
[(tab)commands] [#commentary]

Здесь знак # определяет начало комментария (содержимое от знака # и до конца строки не будет обрабатываться. Одинарное двоеточие указывает на то, что последовательность команд должна содержаться в одной строке. Для переноса можно в длинной строке команд можно использовать обратный слэш (\). Двойное двоеточие указывает на то, что последовательность команд может содержаться в нескольких последовательных строках. Пример более сложного синтаксиса Makefile:

#  
# Makefile for abcd.c  
#  
CC = gcc  
CFLAGS =  
# Compile abcd.c normaly  
abcd: abcd.c  
$(CC) -o abcd $(CFLAGS) abcd.c  
clean:  
-rm abcd \*.o \*~  
# End Makefile for abcd.c

В этом примере в начале файла заданы три переменные: CC и CFLAGS. Затем указаны цели, их зависимости и соответствующие команды. В командах происходит обращение к значениям переменных. Цель с именем clean производит очистку каталога от файлов, полученных в результате компиляции. Для её описания использованы регулярные выражения. 7. Во время работы над кодом программы программист неизбежно сталкивается с появлением ошибок в ней. Использование отладчика для поиска и устранения ошибок в программе существенно облегчает жизнь программиста. В комплект программ GNU для ОС типа UNIX входит отладчик GDB (GNU Debugger). Для использования GDB необходимо скомпилировать анализируемый код программы таким образом, чтобы отладочная информация содержалась в результирующем бинарном файле. Для этого следует воспользоваться опцией -g компилятора gcc: gcc -c file.c -g После этого для начала работы с gdb необходимо в командной строке ввести одноимённую команду, указав в качестве аргумента анализируемый бинарный файл: gdb file.o  
8. Основные команды отладчика gdb: - backtrace − вывод на экран пути к текущей точке останова (по сути вывод − названий всех функций) - break − установить точку останова (в качестве параметра может быть указан номер строки или название функции) - clear − удалить все точки останова в функции - continue − продолжить выполнение программы - delete − удалить точку останова - display − добавить выражение в список выражений, значения которых отображаются при достижении точки останова программы - finish − выполнить программу до момента выхода из функции - info breakpoints − вывести на экран список используемых точек останова - info watchpoints − вывести на экран список используемых контрольных выражений - list − вывести на экран исходный код (в качестве параметра может быть указано название файла и через двоеточие номера начальной и конечной строк) - next − выполнить программу пошагово, но без выполнения вызываемых в программе функций - print − вывести значение указываемого в качестве параметра выражения - run − запуск программы на выполнение - set − установить новое значение переменной - step − пошаговое выполнение программы - watch − установить контрольное выражение, при изменении значения которого программа будет остановлена Для выхода из gdb можно воспользоваться командой quit (или её сокращённым вариантом q) или комбинацией клавиш Ctrl-d. Более подробную информацию по работе с gdb можно получить с помощью команд gdb -h и man gdb. 9. Cхема отладки программы показана в 6 пункте лабораторной работы. 10. При первом запуске компилятор не выдал никаких ошибок, но в коде программы main.c допущена ошибка, которую компилятор мог пропустить (возможно, из-за версии 8.3.0-19): в строке scanf(“%s”, &Operation); нужно убрать знак &, потому что имя массива символов уже является указателем на первый элемент этого массива. 11. Система разработки приложений UNIX предоставляет различные средства, повышающие понимание исходного кода. К ним относятся: - cscope − исследование функций, содержащихся в программе, - lint − критическая проверка программ, написанных на языке Си. 12. Утилита splint анализирует программный код, проверяет корректность задания аргументов использованных в программе функций и типов возвращаемых значений, обнаруживает синтаксические и семантические ошибки. В отличие от компилятора C анализатор splint генерирует комментарии с описанием разбора кода программы и осуществляет общий контроль, обнаруживая такие ошибки, как одинаковые объекты, определённые в разных файлах, или объекты, чьи значения не используются в работе программы, переменные с некорректно заданными значениями и типами и многое другое.

# Вывод

В ходе выполнения лабораторной работы я приобрёл простейшие навыки разработки, анализа, тестирования и отладки приложений в ОС типа UNIX/Linux на примере создания на языке программирования С калькулятора с простейшими функциями.