

Шаблон отчёта по лабораторной работе номер 14

Дисциплина: Операционные системы

Крестененко Полина Александровна

Содержание

1	Цель работы	5
2	Задание	6
3	Выполнение лабораторной работы	7
4	Выводы	19

Список таблиц

Список иллюстраций

3.1	создание подкаталога	7
3.2	создание файлов	7
3.3	первый файл	8
3.4	первый файл	8
3.5	второй файл	9
3.6	третий файл	9
3.7	компиляция программы	9
3.8	makefile	10
3.9	makefile исправленный	11
3.10	удаление и компиляция	11
3.11	отладка программы	12
3.12	run list	12
3.13	list 12,15	12
3.14	list calculate.c:20,29	13
3.15	list calculate.c:20,27	13
3.16	info breakpoints	13
3.17	проверка	14
3.18	print Numeral	14
3.19	display Numeral	14
3.20	info breakpoints and delete 1	14
3.21	установка	15

1 Цель работы

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.

2 Задание

Выполнить задания, представленные в тескте файла лабораторной работы.

3 Выполнение лабораторной работы

- 1) В домашнем каталоге создаю подкаталог `~/work/os/lab_prog` с помощью команды «`mkdir -p ~/work/os/lab_prog`»(рис. -fig. 3.1).

```
[pakrestenenko@pakrestenenko ~]$ mkdir -p ~/work/os/lab_prog
```

Рис. 3.1: создание подкаталога

- 2) Создала в каталоге файлы: `calculate.h`, `calculate.c`, `main.c`, используя команды «`cd ~/work/os/lab_prog`» и «`touch calculate.h calculate.c main.c`»(рис. -fig. 3.2).

```
[pakrestenenko@pakrestenenko ~]$ cd ~/work/os/lab_prog
[pakrestenenko@pakrestenenko lab_prog]$ touch calculate.h calculate.c main.c
[pakrestenenko@pakrestenenko lab_prog]$ ls
calculate.c calculate.h main.c
```

Рис. 3.2: создание файлов

Это будет примитивнейший калькулятор, способный складывать, вычитать, умножать и делить, возводить число в степень, брать квадратный корень, вычислять `sin`, `cos`, `tan`. При запуске он будет запрашивать первое число, операцию, второе число. После этого программа выведет результат и остановится. Открыв редактор Emacs, приступила к редактированию созданных файлов. Реализация функций калькулятора в файле `calculate.c`(рис. -fig. 3.3).

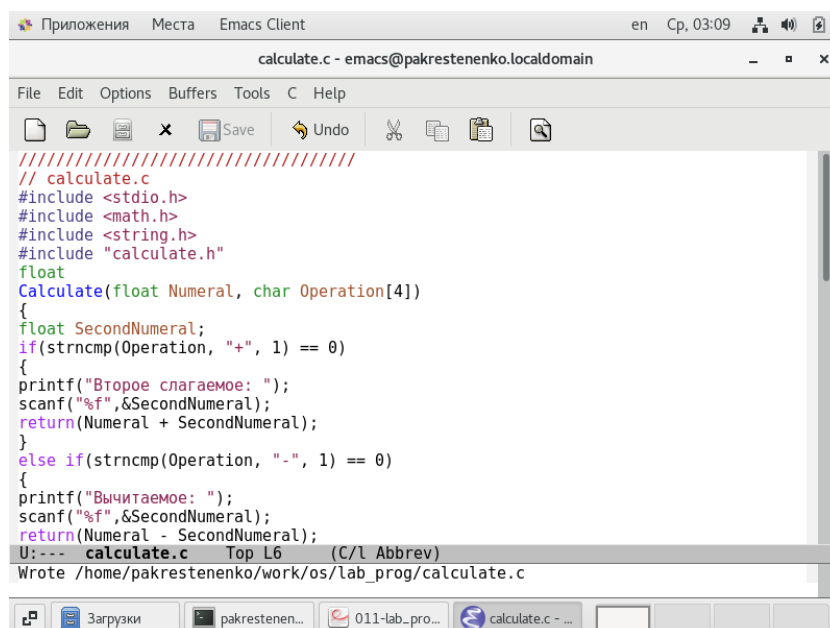


Рис. 3.3: первый файл

(рис. -fig. 3.4).

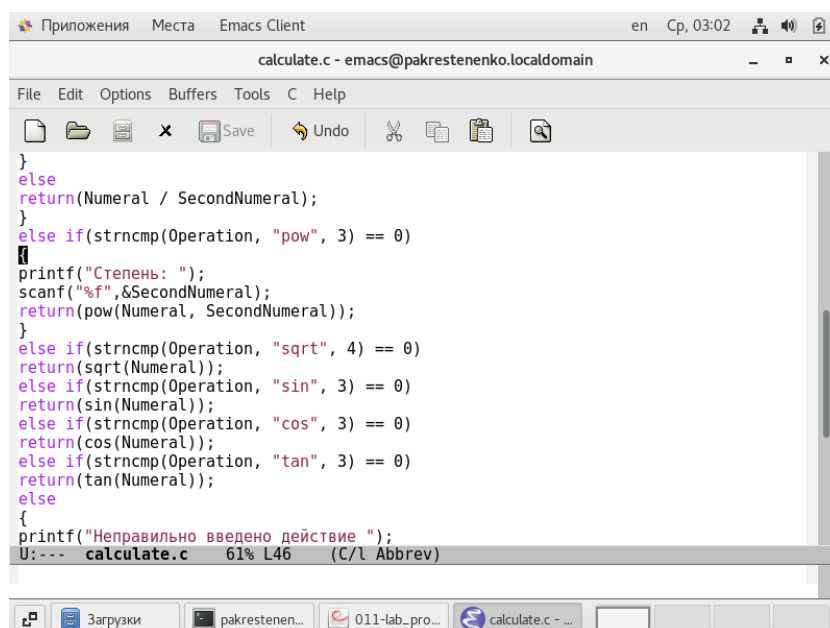
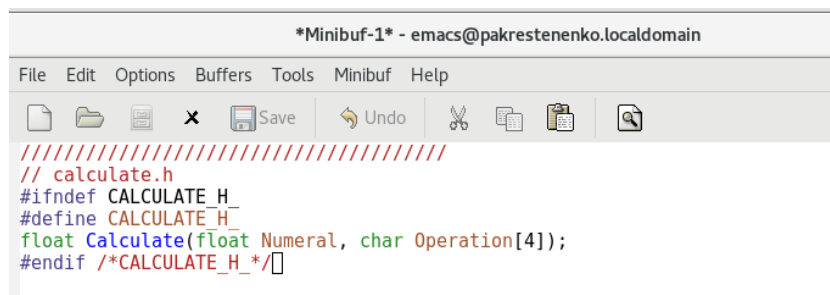


Рис. 3.4: первый файл

Интерфейсный файл calculate.h, описывающий формат вызова функции калькулятора(рис. -fig. 3.5).



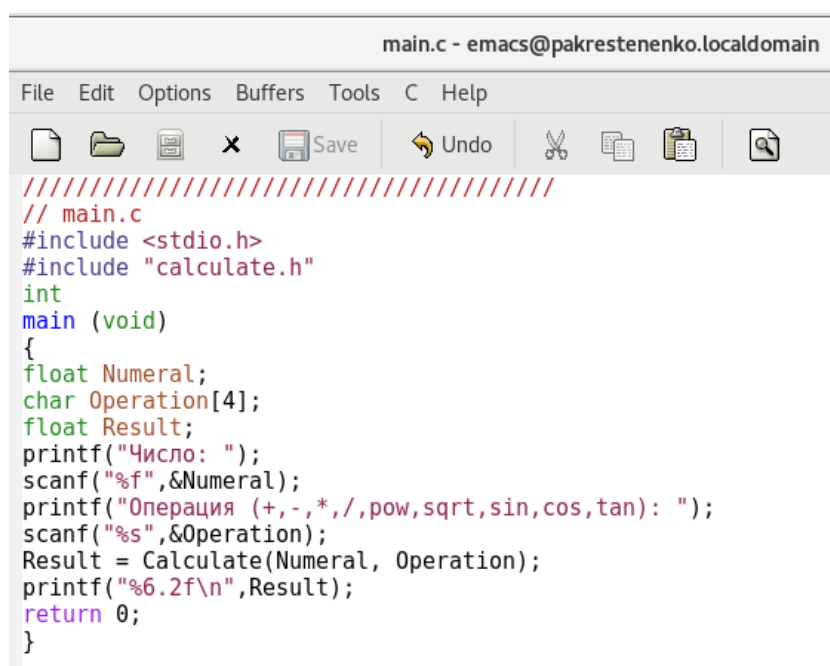
```

*Minibuf-1* - emacs@pakrestenenko.localdomain
File Edit Options Buffers Tools Minibuf Help
Save Undo
// calculate.h
#ifndef CALCULATE_H_
#define CALCULATE_H_
float Calculate(float Numeral, char Operation[4]);
#endif /*CALCULATE_H_*/

```

Рис. 3.5: второй файл

Основной файл main.c, реализующий интерфейс пользователя к калькулятору(рис. -fig. 3.6).



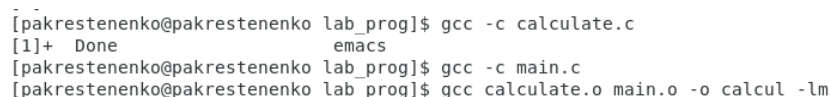
```

main.c - emacs@pakrestenenko.localdomain
File Edit Options Buffers Tools C Help
Save Undo
// main.c
#include <stdio.h>
#include "calculate.h"
int
main (void)
{
float Numeral;
char Operation[4];
float Result;
printf("Число: ");
scanf("%f",&Numeral);
printf("Операция (+, -, *, /, pow, sqrt, sin, cos, tan): ");
scanf("%s",&Operation);
Result = Calculate(Numeral, Operation);
printf("%6.2f\n",Result);
return 0;
}

```

Рис. 3.6: третий файл

- 3) Выполнила компиляцию программы посредством gcc (версия компилятора: 8.3.0-19), используя команды «gcc -c calculate.c», «gcc -c main.c» и «gcc calculate.o main.o -o calcul -lm»(рис. -fig. 3.7).



```

[pakrestenenko@pakrestenenko lab_prog]$ gcc -c calculate.c
[1]+  Done                  emacs
[pakrestenenko@pakrestenenko lab_prog]$ gcc -c main.c
[pakrestenenko@pakrestenenko lab_prog]$ gcc calculate.o main.o -o calcul -lm

```

Рис. 3.7: компиляция программы

- 4) В ходе компиляции программы убрала ошибку в объявлении библиотек
- 5) Создала Makefile с необходимым содержанием (рис. -fig. 3.8).

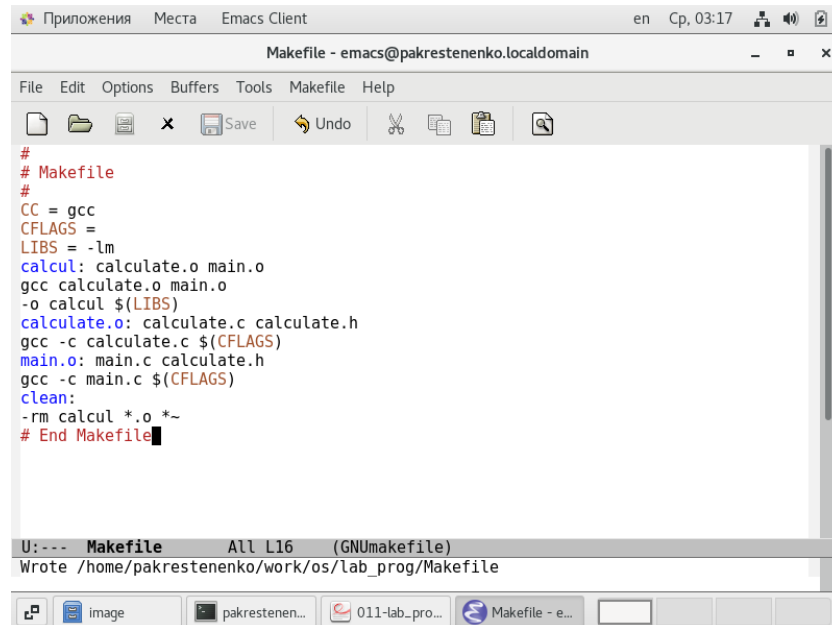


Рис. 3.8: makefile

Данный файл необходим для автоматической компиляции файлов calculate.c (цель calculate.o), main.c (цель main.o), а также их объединения в один исполняемый файл calcul (цель calcul). Цель clean нужна для автоматического удаления файлов. Переменная CC отвечает за утилиту для компиляции. Переменная CFLAGS отвечает за опции в данной утилите. Переменная LIBS отвечает за опции для объединения объектных файлов в один исполняемый файл.

- 6) Далее исправила Makefile (рис. -fig. 3.9).

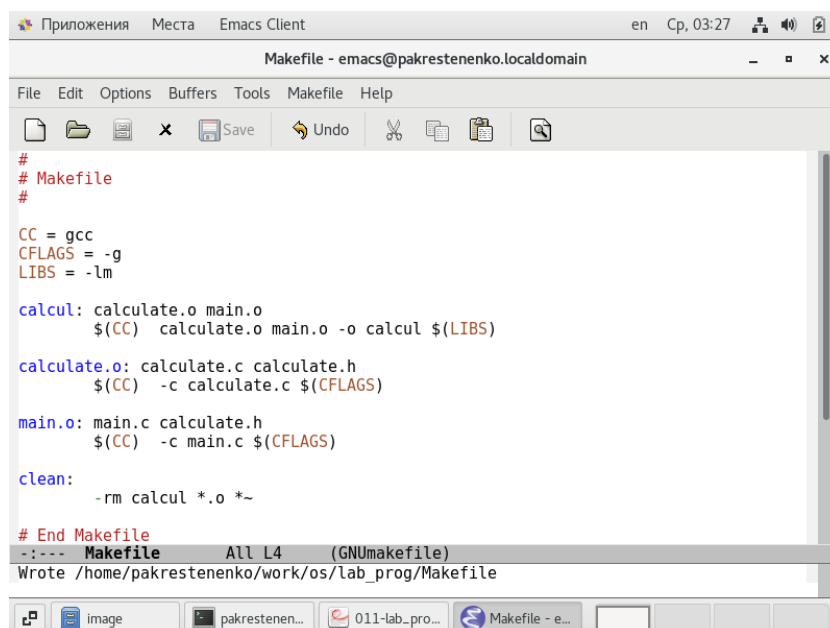


Рис. 3.9: makefile исправленный

В переменную CFLAGS добавила опцию -g, необходимую для компиляции объектных файлов и их использования в программе отладчика GDB. Сделала так, что утилита компиляции выбирается с помощью переменной CC. После этого я удалила исполняемые и объектные файлы из каталога с помощью команды «make clear». Выполнила компиляцию файлов, используя команды «make calculate.o», «make main.o», «make calcul». (рис. -fig. 3.10).

```

[pakrestenenko@pakrestenenko lab_prog]$ make clean
rm calcul *.o *~
[pakrestenenko@pakrestenenko lab_prog]$ make calculate.o
gcc -c calculate.c -g
[pakrestenenko@pakrestenenko lab_prog]$ make main.o
gcc -c main.c -g
[pakrestenenko@pakrestenenko lab_prog]$ make calcul
gcc calculate.o main.o -o calcul -lm

```

Рис. 3.10: удаление и компиляция

Далее с помощью gdb выполнила отладку программы calcul. Запустила отладчик GDB, загрузив в него программу для отладки, используя команду: «gdb ./calcul» (рис. -fig. 3.11).

```
[pakrestenenko@pakrestenenko lab_prog]$ gdb ./calcul
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-120.el7
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/pakrestenenko/work/os/lab_prog/calcul...done.
```

Рис. 3.11: отладка программы

Для запуска программы внутри отладчика ввела команду «run» Для постраничного (по 10 строк) просмотра исходного кода использовала команду «list» (рис. -fig. 3.12).

```
(gdb) run
Starting program: /home/pakrestenenko/work/os/lab_prog/./calcul
Число: 6
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): +
Второе слагаемое: 5
11.00
[Inferior 1 (process 8361) exited normally]
Missing separate debuginfos, use: debuginfo-install glibc-2.17-317.el7.x86_64
(gdb) list
2      // main.c
3      #include <stdio.h>
4      #include "calculate.h"
5      int
6      main (void)
7      {
8      float Numeral;
9      char Operation[4];
10     float Result;
11     printf("Число: ");
(gdb) █
```

Рис. 3.12: run list

Для просмотра строк с 12 по 15 основного файла использовала команду «list 12,15» (рис. -fig. 3.13).

```
(gdb) list 12,15
12     scanf("%f",&Numeral);
13     printf("Операция (+,-,*,/,pow,sqrt,sin,cos,tan): ");
14     scanf("%s",&Operation);
15     Result = Calculate(Numeral, Operation);
```

Рис. 3.13: list 12,15

Для просмотра определённых строк не основного файла использовала команду «list calculate.c:20,29»(рис. -fig. 3.14).

```

(gdb) list calculate.c:20,29
20     scanf("%f",&SecondNumeral);
21     return(Numeral - SecondNumeral);
22 }
23 else if(strncmp(Operation, "*", 1) == 0)
24 {
25     printf("Множитель: ");
26     scanf("%f",&SecondNumeral);
27     return(Numeral * SecondNumeral);
28 }
29 else if(strncmp(Operation, "/", 1) == 0)

```

Рис. 3.14: list calculate.c:20,29

Установила точку останова в файле calculate.c на строке номер 21, используя команды «list calculate.c:20,27» и «break 21»(рис. -fig. 3.15).

```

(gdb) list calculate.c:20,27
20     scanf("%f",&SecondNumeral);
21     return(Numeral - SecondNumeral);
22 }
23 else if(strncmp(Operation, "*", 1) == 0)
24 {
25     printf("Множитель: ");
26     scanf("%f",&SecondNumeral);
27     return(Numeral * SecondNumeral);
(gdb) break 21
Breakpoint 1 at 0x4007fd: file calculate.c, line 21.

```

Рис. 3.15: list calculate.c:20,27

Вывела информацию об имеющихся в проекте точках останова с помощью команды «info breakpoints» (рис. -fig. 3.16).

```

(gdb) info breakpoints
Num   Type             Disp Enb Address                  What
1     breakpoint       keep y   0x00000000004007fd in Calculate at calculate.c:21

```

Рис. 3.16: info breakpoints

Запустила программу внутри отладчика и убедилась, что программа остановилась в момент прохождения точки останова. Использовала команды «run», «5», «-» и «backtrace» (рис. -fig. 3.17).

```

(gdb) run
Starting program: /home/pakrestenenko/work/os/lab_prog/./calcul
Число: 5
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): -
Вычитаемое: 1

Breakpoint 1, Calculate (Numeral=5, Operation=0x7fffffffdea0 "-") at calculate.c:21
21      return(Numeral - SecondNumeral);
(gdb) print Numeral
$1 = 5

```

Рис. 3.17: проверка

Посмотрела, чему равно на этом этапе значение переменной Numeral, введя команду «print Numeral» (рис. -fig. 3.18).

```

(gdb) print Numeral
$1 = 5
(gdb) display Numeral

```

Рис. 3.18: print Numeral

Сравнила с результатом вывода на экран после использования команды «display Numeral». Значения совпадают(рис. -fig. 3.19).

```

(gdb) display Numeral
1: Numeral = 5

```

Рис. 3.19: display Numeral

Убрала точки останова с помощью команд «info breakpoints» и «delete 1»(рис. -fig. 3.20).

```

(gdb) info breakpoints
Num   Type             Disp Enb Address            What
1      breakpoint       keep y   0x00000000004007fd in Calculate at calculate.c:21
      breakpoint already hit 1 time
(gdb) delete 1

```

Рис. 3.20: info breakpoints and delete 1

7) С помощью утилиты splint проанализировала коды файлов calculate.c и main.c. Предварительно я установила данную утилиту с помощью команд

«sudo apt update» и «sudo apt install splint» Далее воспользовалась командами «splint calculate.c» и «splint main.c».

(рис. -fig. 3.21)

```
[pakrestenenko@pakrestenenko lab_prog]$ sudo apt update
```

Рис. 3.21: установка

С помощью утилиты splint выяснилось, что в файлах calculate.c и main.c присутствует функция чтения scanf, возвращающая целое число (тип int), но эти числа не используются и нигде не сохраняются. Утилита вывела предупреждение о том, что в файле calculate.c происходит сравнение вещественного числа с нулем. Также возвращаемые значения (тип double) в функциях pow, sqrt, sin, cos и tan записываются в переменную типа float, что свидетельствует о потере данных.

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: о кодирование – по сути создание исходного текста программы (возможно в нескольких вариантах); – анализ разработанного кода; о сборка, компиляция и разработка исполняемого модуля; о 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. 3) Для имени входного файла суффикс определяет какая компиляция требуется. Суффиксы указывают на тип объекта. Файлы с расшире-

нием (суффиксом) .c воспринимаются gcc как программы на языке C, файлы с расширением .cc или .C – как файлы на языке C++, а файлы с 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».

4) Основное назначение компилятора языка Си в UNIX заключается в компиляции всей программы и получении исполняемого файла/модуля. 5) 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. 6) 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: ... : ... <команда 1> ... 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. 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 normally 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) Схема 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.

4 Выводы

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