

Отчет по лабораторной работе №12

Средства, применяемые при разработке программного
обеспечения в ОС типа UNIX/Linux

Сомсиков Даниил Сергеевич

Содержание

1	Цель работы	4
2	Задание	5
3	Теоретическое введение	6
4	Выполнение лабораторной работы	7
5	Контрольные вопросы	16
6	Выводы	22
	Список литературы	23

Список иллюстраций

4.1 Создание нового подкаталога и файлов	7
4.2 calculate.h	8
4.3 calculate.c	8
4.4 main.c	9
4.5 компиляция программы посредством gcc	9
4.6 Makefile	9
4.7 Запуск отладчика	11
4.8 Просмотр кода и точка останова	12
4.9 Проверка останова и удаление точки останова	13
4.10 Анализ кода файла main.c	14
4.11 Анализ кода файла calculate.c	15

1 Цель работы

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

2 Задание

1. В домашнем каталоге создайте подкаталог `~/work/os/lab_prog`.
2. Создайте в нём файлы: `calculate.h`, `calculate.c`, `main.c`. 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.
3. Выполните компиляцию программы посредством `gcc`.
4. Создайте `Makefile` и поясните о его содержании.
5. С помощью `gdb` выполните отладку программы `calcul` (перед использованием `gdb` исправьте `Makefile`)
6. С помощью утилиты `splint` попробуйте проанализировать коды файлов `calculate.c` и `main.c`.

3 Теоретическое введение

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.

4 Выполнение лабораторной работы

1. В домашнем каталоге создаем новый подкаталог `~/work/os/lab_prog`, переходим в него и создаем 3 файла: `calculate.h`, `calculate.c`, `main.c` (рис. 4.1):

```
dssomsikov@dssomsikov:~$ mkdir work/os/lab_prog | cd work/os/lab_prog
bash: cd: work/os/lab_prog: Нет такого файла или каталога
dssomsikov@dssomsikov:~$ ls
abcl      cprog      may         prog11-1.sh  prog10-3.sh  reports      Изображения
archive.tar feathers    monthly     prog11-2.sh  prog10-4.sh  ski.places   Музыка
australia Files.txt  my_os       prog11-3.sh  program1.sh  work         Общедоступные
backup    file.txt   output.txt  prog10-1.sh  program2.sh  Видео       'Рабочий стол'
Catalog   input.txt  pandoc-crossref prog10-2.c   program3.sh  Документы   Шаблоны
conf.txt   lock.file  play        prog10-2.sh  program.sh    Загрузки

dssomsikov@dssomsikov:~$ cd work/os/
dssomsikov@dssomsikov:~/work/os$ ls
lab06  lab_prog
dssomsikov@dssomsikov:~/work/os$ lab_prog/
bash: lab_prog/: Это каталог
dssomsikov@dssomsikov:~/work/os$ cd lab_prog/
dssomsikov@dssomsikov:~/work/os/lab_prog$ touch cacl.h calc.c main.c
dssomsikov@dssomsikov:~/work/os/lab_prog$ ls
cacl.h  calc.c  main.c
dssomsikov@dssomsikov:~/work/os/lab_prog$
```

Рис. 4.1: Создание нового подкаталога и файлов

2. Запишем в файлы тексты программ, которые даны в руководстве к лабораторной работе (рис. 4.2), (рис. 4.3), (рис. 4.4).

```
// calculate.h

#ifndef CALCULATE_H_
#define CALCULATE_H_

float Calculate(float Numeral, char Operation[4]);

#endif /* CALCULATE_H_ */
```

Рис. 4.2: calculate.h

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include "calculate.h"

float Calculate(float Numeral, char Operation[4]) {
    float SecondNumeral;
    if (strcmp(Operation, "+") == 0) {
        printf("Второе слагаемое: ");
        scanf("%f", &SecondNumeral);
        return (Numeral + SecondNumeral);
    } else if (strcmp(Operation, "-") == 0) {
        printf("Вычитаемое: ");
        scanf("%f", &SecondNumeral);
        return (Numeral - SecondNumeral);
    } else if (strcmp(Operation, "*") == 0) {
        printf("Множитель: ");
        scanf("%f", &SecondNumeral);
        return (Numeral * SecondNumeral);
    } else if (strcmp(Operation, "/") == 0) {
        printf("Делитель: ");
        scanf("%f", &SecondNumeral);
        if (SecondNumeral == 0) {
            printf("Ошибка: деление на ноль! ");
            return (HUGE_VAL);
        } else {
            return (Numeral / SecondNumeral);
        }
    } else if (strcmp(Operation, "pow", 3) == 0) {
        printf("Степень: ");
        scanf("%f", &SecondNumeral);
        return (pow(Numeral, SecondNumeral));
    }
    return 0;
}
```

Рис. 4.3: calculate.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("%06.2f\n", Result);

    return 0;
}

```

Рис. 4.4: main.c

3. Выполним компиляцию программы посредством gcc (рис. 4.5):

```

dssomsikov@dssomsikov:~/work/os/lab_prog$ gcc -c calculate.c
dssomsikov@dssomsikov:~/work/os/lab_prog$ 
dssomsikov@dssomsikov:~/work/os/lab_prog$ gcc -c main.c
dssomsikov@dssomsikov:~/work/os/lab_prog$ gcc calculate.o main.o -o calcul -lm

```

Рис. 4.5: компиляция программы посредством gcc

4. Создадим Makefile (рис. 4.6):

```

# Makefile

CC = gcc
CFLAGS = -g
LIBS = -lm

calcul: calculate.o main.o
    $(CC) calculate.o main.o -o calcul $(LIBS)

calculate.o: calculate.c calculate.h
    $(CC) -c calculate.c $(CFLAGS)

main.o: main.c calculate.h
    $(CC) -c main.c $(CFLAGS)

clean:
    rm -f calcul *.o *~

# End Makefile

```

Рис. 4.6: Makefile

Этот Makefile используется для автоматизации процесса компиляции и сборки программы.

Цели и правила в Makefile:

- `calcul`: Эта цель отвечает за создание исполняемого файла `calcul`. Она зависит от объектных файлов `calculate.o` и `main.o`. Команда `$(CC) calculate.o main.o -o calcul $(LIBS)` компилирует эти объектные файлы вместе с библиотеками, указанными в переменной `LIBS`, для создания исполняемого файла.
- `calculate.o`: Это правило указывает `make`, как создать объектный файл `calculate.o` из исходного файла `calculate.c` и заголовочного файла `calculate.h`. Команда `$(CC) -c calculate.c $(CFLAGS)` компилирует исходный файл `calculate.c` в объектный файл, используя флаги компиляции, заданные в переменной `CFLAGS`.
- `main.o`: Это правило аналогично предыдущему, но предназначено для создания объектного файла `main.o` из исходного файла `main.c` и заголовочного файла `calculate.h`.
- `clean`: Эта специальная цель предназначена для очистки каталога от файлов, созданных в процессе сборки. Команда `rm calcul *.o *` удаляет исполняемый файл `calcul`, все объектные файлы (`*.o`) и временные файлы, созданные редакторами (файлы, оканчивающиеся на `~`).

Знак минуса (-) перед командой `rm` указывает `make` игнорировать ошибки при удалении файлов (например, если файл уже был удалён).

5. С помощью `gdb` выполним отладку программы `calcul` (я привела код уже исправленного Makefile: ошибка заключалась в отсутствии опции `-g` у `CFLAG`):

- Запустим отладчик GDB, загрузив в него программу для отладки, используя `gdb ./calcul`. Для запуска программы внутри отладчика введем команду `run` и посчитаем некое выражение (рис. 4.7)

```
dssomsikov@dssomsikov:~/work/os/lab_prog$ gdb ./calcul
GNU gdb (Fedora Linux) 14.2-2.fc40
Copyright (C) 2023 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".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./calcul...

This GDB supports auto-downloading debuginfo from the following URLs:
<https://debuginfod.fedoraproject.org/>
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
Downloading separate debug info for /home/dssomsikov/work/os/lab_prog/calcul
(No debugging symbols found in ./calcul)
(gdb) run
Starting program: /home/dssomsikov/work/os/lab_prog/calcul
Downloading separate debug info for system-supplied DSO at 0x7ffff7fc7000
Downloading separate debug info for /lib64/libm.so.6
Downloading separate debug info for /lib64/libc.so.6
warning: `'/home/dssomsikov/.cache/debuginfod_client/4a92fcedbba6d6d2629ce066a2970017faa9995e/debuginfo': can't read s
ymbols: формат файла не распознан.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Число: 40
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): *
Множитель: 10
400.00
[Inferior 1 (process 27467) exited normally]
(gdb) █
```

Рис. 4.7: Запуск отладчика

- Для постраничного (по 9 строк) просмотра исходного кода используем команду `list`, затем для просмотра строк с 12 по 15 основного файла используем `list 12,15`, просмотрим определённых строк не основного файла, используя `list calculate.c:20,29`, а также установим точку останова в файле `calculate.c` на строке номер 21, используя `list calculate.c:20,27` и `break 21` (рис. 4.8):

```

(gdb) list
1      #include <stdio.h>
2      #include "calculate.h"
3
4      int main(void) {
5          float Numeral;
6          char Operation[4];
7          float Result;
8
9          printf("Число: ");
10         scanf("%f", &Numeral);
(gdb) list 12,15
12         scanf("%s", Operation);
13
14         Result = Calculate(Numeral, Operation);
15         printf("%06.2f\n", Result);
(gdb) list calculate.c:20,29
20     } else if (strcmp(Operation, "/") == 0) {
21         printf("Делитель: ");
22         scanf("%f", &SecondNumeral);
23         if (SecondNumeral == 0) {
24             printf("Ошибка: деление на ноль! ");
25             return (HUGE_VAL);
26         } else {
27             return (Numeral / SecondNumeral);
28         }
29     } else if (strcmp(Operation, "pow", 3) == 0) {
(gdb)

```

Рис. 4.8: Просмотр кода и точка останова

Запустим программу внутри отладчика с помощью `run` и убедимся, что программа остановится в момент прохождения точки останова. С помощью команды `backtrace` покажем весь стек вызываемых функций от начала программы до текущего места. Посмотрим, чему равно на этом этапе значение переменной `Numeral`, введя `print Numeral` и сравним с результатом вывода на экран после использования команды, использовав `display Numeral`. Посмотрим, информацию про точку останова с помощью `info breakpoints` и удалим эту точку командой `delete 1` (рис. 4.9):

```

(gdb) list calculate.c:14,18
14         scanf("%f", &SecondNumeral);
15         return (Numeral - SecondNumeral);
16     } else if (strncmp(Operation, "+", 1) == 0) {
17         printf("Множитель: ");
18         scanf("%f", &SecondNumeral);
(gdb) break 14
Note: breakpoint 1 also set at pc 0x4011de.
Breakpoint 2 at 0x4011de: file calculate.c, line 14.
(gdb) run
Starting program: /home/dssomsikov/work/os/lab_prog/calcul
warning: `/home/dssomsikov/.cache/debuginfod_client/4a92fcedbba6d6d2629ce066a2970017faa9995e/debuginfo': can't read s
ymbols: формат файла не распознан.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Число: 20
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): -

Breakpoint 1, Calculate (Numeral=20, Operation=0x7fffffffddde4 "-") at calculate.c:14
14         scanf("%f", &SecondNumeral);
(gdb) backtrace
#0 Calculate (Numeral=20, Operation=0x7fffffffddde4 "-") at calculate.c:14
#1 0x00000000004013a9 in main () at main.c:14
(gdb) print Numeral
$1 = 20
(gdb) display Numeral
1: Numeral = 20
(gdb) info breakpoints
Num      Type      Disp Enb Address          What
1        breakpoint keep y 0x00000000004011de in Calculate at calculate.c:14
         breakpoint already hit 1 time
2        breakpoint keep y 0x00000000004011de in Calculate at calculate.c:14
         breakpoint already hit 1 time
(gdb) delete 1
(gdb) delete 2
(gdb)

```

Рис. 4.9: Проверка остановки и удаление точки останова

6. С помощью утилиты `splint` попробуйте проанализировать коды файлов `main.c` и `calculate.c`.

Разберем сначала файл `main.c`. Сначала выводится информация о том, что длина массива, указанная в сигнатуре функции `calculate`, не имеет никакого смысла и игнорируется. Далее несколько раз выводится информация о том что мы игнорируем возвращаемое значение функции `scanf`. Всего 3 предупреждения (рис. 4.10).

```

dssomsikov@dssomsikov:~/work/os/lab_prog$ splint main.c
bash: splint: команда не найдена...
Установить пакет «splint», предоставляющий команду «splint»? [N/y] y

* Ожидание в очереди...
* Загрузка списка пакетов....
Следующие пакеты должны быть установлены:
splint-3.1.2-31.fc39.x86_64    An implementation of the lint program
Продолжить с этими изменениями? [N/y] y

* Ожидание в очереди...
* Ожидание аутентификации...
* Ожидание в очереди...
* Загрузка пакетов...
* Запрос данных...
* Проверка изменений...
* Установка пакетов...
Splint 3.1.2 --- 22 Jul 2023

calculate.h:6:37: Function parameter Operation declared as manifest array (size
        constant is meaningless)
  A formal parameter is declared as an array with size.  The size of the array
  is ignored in this context, since the array formal parameter is treated as a
  pointer. (Use -fixedformalarray to inhibit warning)
main.c: (in function main)
main.c:10:5: Return value (type int) ignored: scanf("%f", &Num...
  Result returned by function call is not used. If this is intended, can cast
  result to (void) to eliminate message. (Use -retvalint to inhibit warning)
main.c:12:5: Return value (type int) ignored: scanf("%s", Oper...

Finished checking --- 3 code warnings
dssomsikov@dssomsikov:~/work/os/lab_prog$ █

```

Рис. 4.10: Анализ кода файла main.c

Теперь разберем файл calculate.c. Те же предупреждения о игнорировании длины массива и возвращаемого значения функции scanf, а также предупреждения о неявном преобразовании типа double в тип float. В сумме 15 предупреждений.

```

dssomsikov@dssomsikov:~/work/os/lab_prog$ splint calculate.c
Splint 3.1.2 --- 22 Jul 2023

calculate.h:6:37: Function parameter Operation declared as manifest array (size
        constant is meaningless)
    A formal parameter is declared as an array with size. The size of the array
    is ignored in this context, since the array formal parameter is treated as a
    pointer. (Use -fixedformalarray to inhibit warning)
calculate.c:6:37: Function parameter Operation declared as manifest array (size
        constant is meaningless)
calculate.c: (in function Calculate)
calculate.c:10:9: Return value (type int) ignored: scanf("%f", &Sec...
    Result returned by function call is not used. If this is intended, can cast
    result to (void) to eliminate message. (Use -retvalint to inhibit warning)
calculate.c:14:9: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:18:9: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:22:9: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:23:13: Dangerous equality comparison involving float types:
        SecondNumeral == 0
    Two real (float, double, or long double) values are compared directly using
    == or != primitive. This may produce unexpected results since floating point
    representations are inexact. Instead, compare the difference to FLT_EPSILON
    or DBL_EPSILON. (Use -realcompare to inhibit warning)
calculate.c:25:20: Return value type double does not match declared type float:
        (HUGE_VAL)
    To allow all numeric types to match, use +relaxtypes.
calculate.c:31:9: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:32:16: Return value type double does not match declared type float:
        (pow(Numeral, SecondNumeral))

Finished checking --- 10 code warnings
dssomsikov@dssomsikov:~/work/os/lab_prog$

```

Рис. 4.11: Анализ кода файла calculate.c

5 Контрольные вопросы

1. Как получить информацию о возможностях программ `gcc`, `make`, `gdb` и др.?

Можно использовать команду `man имя_программы` или посетить сайт проекта GNU [`gnu_docs?`].

2. Назовите и дайте краткую характеристику основным этапам разработки приложений в UNIX.

- 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.

3. Что такое суффикс в контексте языка программирования? Приведите примеры использования.

В предоставленной теории [yamad?], суффикс эквивалентен расширению файла. Если же мы говорим о языке Си, в нем суффиксом называется то, что дописывается в конце литерала, например: `3.14f`. В этом примере `f` суффикс, который говорит о том, что литерал будет иметь тип `float`.

4. Каково основное назначение компилятора языка C в UNIX?

Основное назначение компилятора языка Си в UNIX состоит в том, чтобы преобразовывать исходный код, написанный на языке программирования Си, в исполняемый файл, который может быть запущен операционной системой UNIX.

5. Для чего предназначена утилита `make`? Утилита `make` является мощным инструментом автоматизации сборки программного обеспечения, который широко используется в UNIX-системах. Ее основное назначение – управление процессом компиляции и сборки программ, обеспечивая эффективное и удобное обновление исполняемых файлов при внесении изменений в исходный код.

6. Приведите пример структуры `Makefile`. Дайте характеристику основным элементам этого файла.

Пример структуры `Makefile` и его характеристику можно увидеть на лист. 5.1.

7. Назовите основное свойство, присущее всем программам отладки. Что необходимо сделать, чтобы его можно было использовать?

Основное свойство, присущее всем программам отладки, – это возможность отслеживать выполнение программы, шаг за шагом, и анализировать ее состояние в любой момент времени. Это свойство называется “отладкой” или “`debug mode`”. Отладка позволяет разработчикам выявлять и исправлять ошибки в

программе, а также понимать логику ее работы. Чтобы использовать это свойство, необходимо выполнить следующие шаги:

1. Включить отладочную информацию при компиляции программы:
Это делается с помощью флагов компилятора, например, `-g` в компиляторах GCC. Флаг `-g` указывает компилятору включить отладочную информацию в объектные файлы и исполняемый файл. Отладочная информация включает в себя данные о символах (таких как имена переменных и функций), строках кода и расположении переменных в памяти.
 2. Использовать отладчик: Отладчик – это программа, которая позволяет взаимодействовать с выполняющейся целью и контролировать ее выполнение. Примерами отладчиков являются GDB (GNU Debugger) для C/C++ программ и `pdb` для Python-программ. Отладчик позволяет устанавливать точки останова, просматривать значения переменных, выполнять код пошагово и изучать стеки вызовов функций.
 3. Компилировать программу с отключенными оптимизациями:
Некоторые оптимизации компилятора могут усложнить процесс отладки.
8. Назовите и дайте основную характеристику основным командам отладчика `gdb`.

Основные команды отладчика GDB (GNU Debugger) включают в себя следующее:

- `break`: устанавливает точку останова в указанной строке кода или функции. Когда программа выполняется и достигает точки останова, она приостанавливает свое выполнение, позволяя вам проанализировать ее состояние.
- `run`: запускает программу под контролем отладчика. Про-

грамма выполняется до первой точки останова или до завершения.

- `continue`: продолжает выполнение программы после остановки в точке останова.
- `print`: выводит значение выражения или переменной. Это позволяет проверять текущие значения переменных во время выполнения программы.
- `backtrace`: отображает стек вызовов функций, показывая последовательность функций, которые были вызваны для достижения текущей точки выполнения. Это помогает понять поток управления в программе.
- `step`: выполняет программу пошагово, переходя к следующей строке кода. Если следующая строка содержит вызов функции, отладчик заходит внутрь этой функции.
- `next`: выполняет программу пошагово, но в отличие от `step`, он переходит к следующей строке кода, не заходя внутрь вызываемых функций.
- `finish`: продолжает выполнение программы до выхода из текущей функции.
- `info breakpoints`: информация о имеющихся точках останова.

9. Опишите по шагам схему отладки программы, которую Вы использовали при выполнении лабораторной работы.

1. Собрать программу с ключем `-g`
2. Загрузить программу в отладчик GDB.
3. Расставить точки останова.
4. Запустить загруженную программу командой `run`.

10. Прокомментируйте реакцию компилятора на синтаксические ошибки в программе при его первом запуске.

К сожалению, я переписал программу калькулятора без них, так как думал что это опечатки, и у меня не возникло никаких ошибок компиляции. Просмотрев код программы из [yamad?], я вижу одну грубую ошибку (файл main.c строка 16): `scanf("%s",&Operation);`, здесь не нужно брать адрес переменной `Operation`, т.к. мы передадим функции `scanf` `char**`, а она ожидает `char*`.

11. Назовите основные средства, повышающие понимание исходного кода программы.

- инструменты статического анализа, линтеры (такие как `splint`)
- современные IDE предлагают различные функции, облегчающие понимание кода, такие как подсветка синтаксиса и автодополнение
- отладчики (такие как GDB)

12. Каковы основные задачи, решаемые программой `splint`? Программа `Splint` предназначена для решения следующих основных задач:

1. Статический анализ кода: `Splint` выполняет статический анализ кода на языке C, выявляя потенциальные ошибки, проблемы безопасности и нарушения стандартов кодирования. Он проверяет код на соответствие определенным правилам и стандартам, таким как правила из руководства по стилю кодирования MISRA C.
2. Выявление ошибок времени компиляции: `Splint` анализирует код на наличие синтаксических и семантических ошибок, которые могут привести к ошибкам во время компиляции. Он проверяет типы переменных, соответствие аргументов

функций и соблюдение правил объявления переменных.

3. Проверка безопасности: Splint специализируется на выявлении потенциальных проблем безопасности в коде, таких как переполнение буфера, использование неинициализированных переменных, ошибки управления памятью и другие распространенные уязвимости. Он помогает разработчикам писать более безопасный и защищенный от атак код.
4. Подсказки по улучшению кода: Splint предоставляет подсказки и рекомендации по улучшению качества кода.

Листинг 5.1 Пример Makefile

```
# Определение переменных
CC = gcc
CFLAGS = -Wall -O2

# Определение цели по умолчанию
all: программа

# Правило для сборки исполняемого файла
программа: программа.o функция.o
    $(CC) $(CFLAGS) -o программа программа.o функция.o

# Правило для компиляции исходного файла в объектный файл
.c.o:
    $(CC) $(CFLAGS) -c $< -o $@

# Правило для удаления объектных файлов и исполняемого файла
clean:
    rm -f программа.o функция.o программа
```

6 Выводы

В данной лабораторной работе мы приобрели простейшие навыки разработки, анализа, тестирования и отладки приложений в ОС типа UNIX/Linux на примере создания на языке программирования C калькулятора с простейшими функциями.

Список литературы

1. Руководство к лабораторной работе №12.