

# **Отчёт лабораторной работы №14**

**Дисциплина: Операционные системы**

Касьянов Даниил Владимирович

# Содержание

1	Цель работы	5
2	Выполнение лабораторной работы	6
3	Контрольные вопросы	18
4	Выводы	24
5	Библиография	25

## Список таблиц

## **Список иллюстраций**

# 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 Выполнение лабораторной работы

1. В домашнем каталоге создаю подкаталог `~/work/os/lab_prog` (Рисунок 1).

```
dvkasjyanov@dvkasjyanov:~$ mkdir -p ~/work/os/lab_prog
```

(Рисунок 1)

2. Создаю в нём файлы **calculate.h**, **calculate.c**, **main.c** (Рисунок 2). 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.

```
dvkasjyanov@dvkasjyanov:~$ cd ~/work/os/lab_prog
dvkasjyanov@dvkasjyanov:~/work/os/lab_prog$ touch calculate.h calculate.c main.c
dvkasjyanov@dvkasjyanov:~/work/os/lab_prog$ ls
calculate.c calculate.h main.c
dvkasjyanov@dvkasjyanov:~/work/os/lab_prog$
```

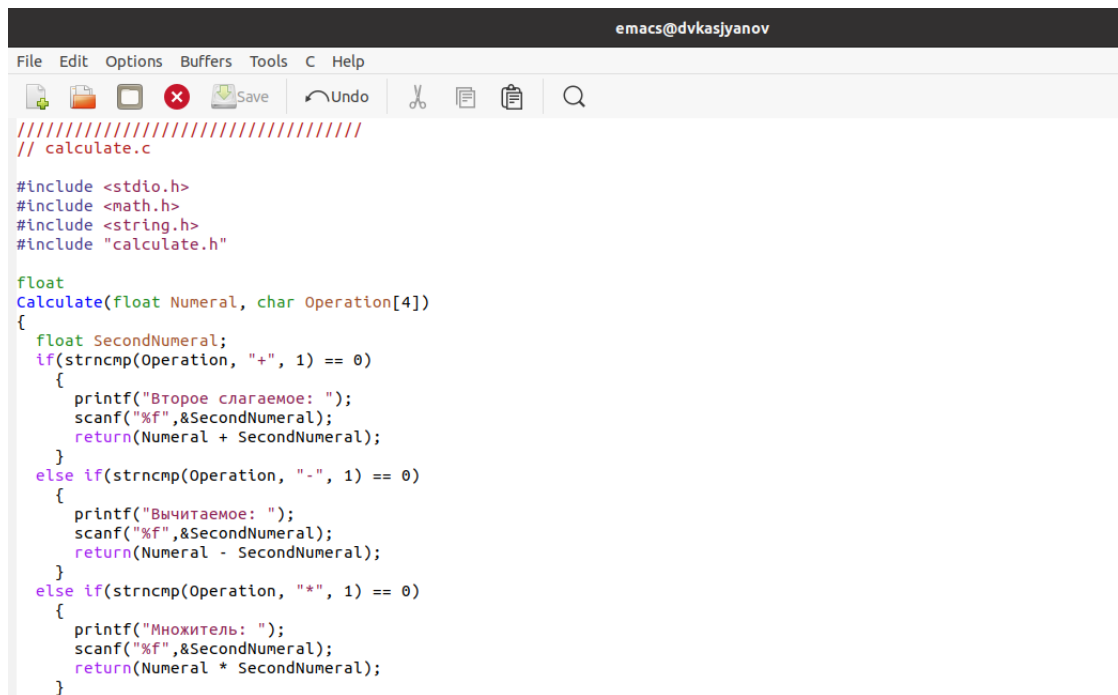
(Рисунок 2)

Скопирую тексты из лабораторной работы и вставлю в файлы.

- **calculate.c** (Рис. 3, 4, 5):

```
dvkasjyanov@dvkasjyanov:~/work/os/lab_prog$ emacs calculate.c
```

(Рисунок 3)

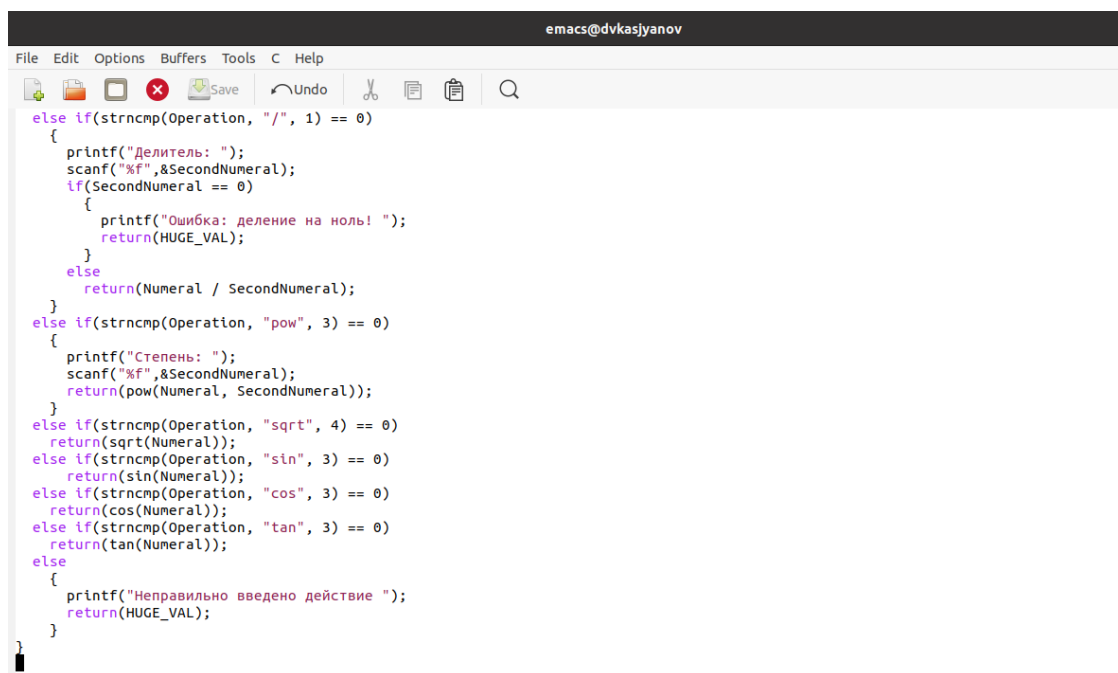


```
emacs@dvmkasjyanov
File Edit Options Buffers Tools C Help
Save Undo
////////////////////////////////////
// calculate.c

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

float
Calculate(float Numeral, char Operation[4])
{
    float SecondNumeral;
    if(strncmp(Operation, "+", 1) == 0)
    {
        printf("Второе слагаемое: ");
        scanf("%f", &SecondNumeral);
        return(Numeral + SecondNumeral);
    }
    else if(strncmp(Operation, "-", 1) == 0)
    {
        printf("Вычитаемое: ");
        scanf("%f", &SecondNumeral);
        return(Numeral - SecondNumeral);
    }
    else if(strncmp(Operation, "*", 1) == 0)
    {
        printf("Множитель: ");
        scanf("%f", &SecondNumeral);
        return(Numeral * SecondNumeral);
    }
}
```

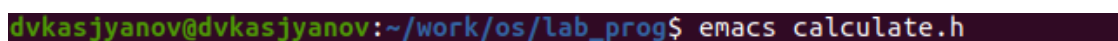
(Рисунок 4)



```
emacs@dvmkasjyanov
File Edit Options Buffers Tools C Help
Save Undo
else if(strncmp(Operation, "/", 1) == 0)
{
    printf("Делитель: ");
    scanf("%f", &SecondNumeral);
    if(SecondNumeral == 0)
    {
        printf("Ошибка: деление на ноль! ");
        return(HUGE_VAL);
    }
    else
        return(Numeral / SecondNumeral);
}
else if(strncmp(Operation, "pow", 3) == 0)
{
    printf("Степень: ");
    scanf("%f", &SecondNumeral);
    return(pow(Numeral, SecondNumeral));
}
else if(strncmp(Operation, "sqrt", 4) == 0)
    return(sqrt(Numeral));
else if(strncmp(Operation, "sin", 3) == 0)
    return(sin(Numeral));
else if(strncmp(Operation, "cos", 3) == 0)
    return(cos(Numeral));
else if(strncmp(Operation, "tan", 3) == 0)
    return(tan(Numeral));
else
{
    printf("Неправильно введено действие ");
    return(HUGE_VAL);
}
}
```

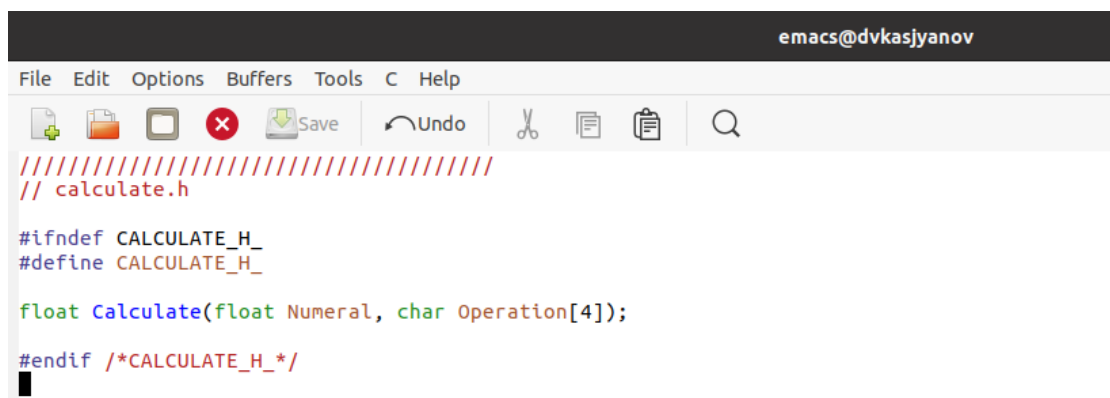
(Рисунок 5)

- **calculate.h** (Рис. 6, 7):



```
dvmkasjyanov@dvmkasjyanov:~/work/os/lab_prog$ emacs calculate.h
```

(Рисунок 6)



```
emacs@dvkasjyanov
File Edit Options Buffers Tools C Help
Save Undo
////////////////////////////////////
// calculate.h

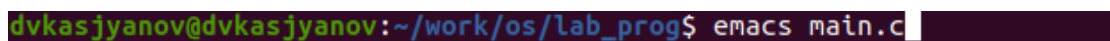
#ifndef CALCULATE_H_
#define CALCULATE_H_

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

#endif /*CALCULATE_H_*/
```

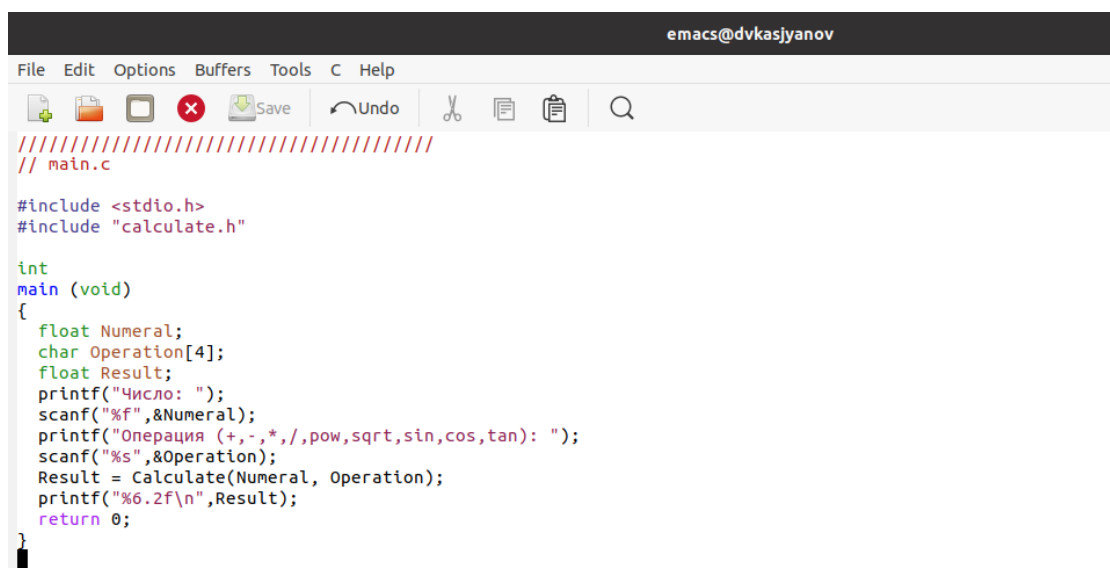
(Рисунок 7)

- **main.c** (Рис. 8, 9):



```
dvkasjyanov@dvkasjyanov:~/work/os/lab_prog$ emacs main.c
```

(Рисунок 8)



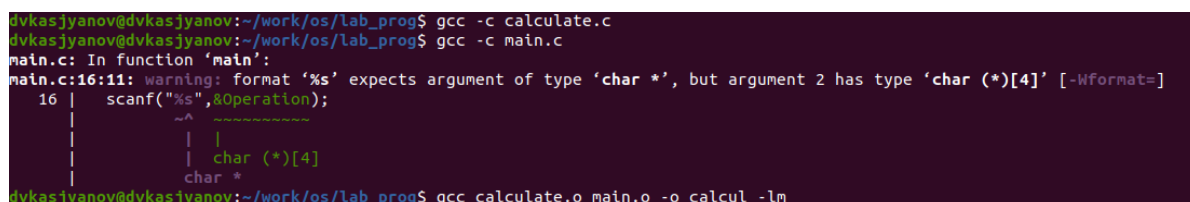
```
emacs@dvkasjyanov
File Edit Options Buffers Tools C Help
Save Undo
////////////////////////////////////
// main.c

#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("%.2f\n",Result);
    return 0;
}
```

(Рисунок 9)

### 3. Выполняю компиляцию программы посредством **gcc** (Рисунок 10).



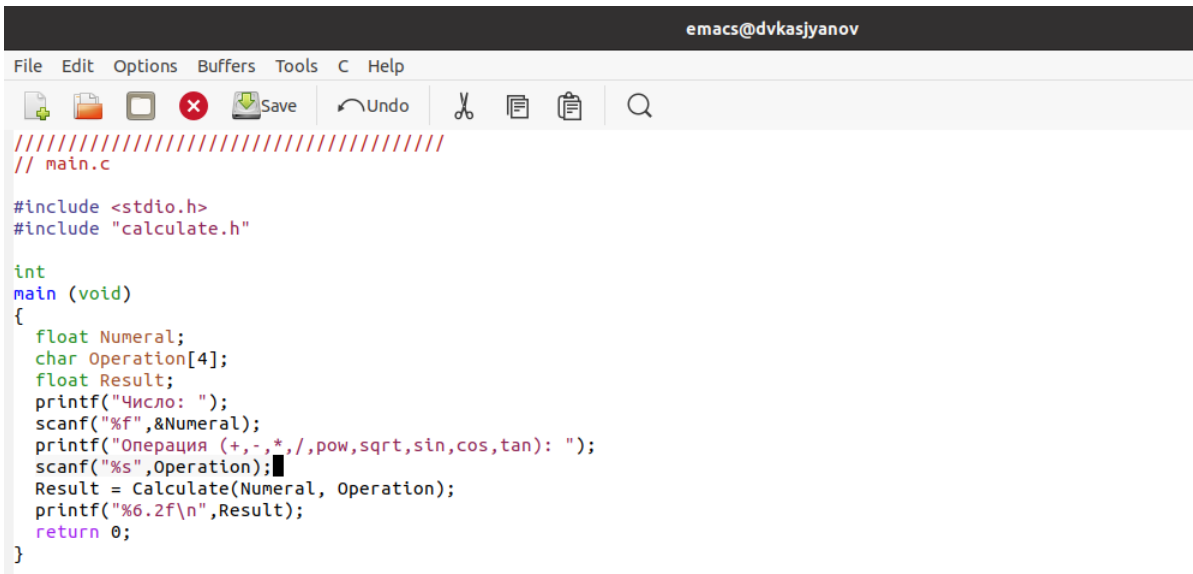
```
dvkasjyanov@dvkasjyanov:~/work/os/lab_prog$ gcc -c calculate.c
dvkasjyanov@dvkasjyanov:~/work/os/lab_prog$ gcc -c main.c
main.c: In function 'main':
main.c:16:11: warning: format '%s' expects argument of type 'char *', but argument 2 has type 'char (*)[4]' [-Wformat=]
   16 |     scanf("%s",&Operation);
      |           ~^ ~~~~~
      |           |   char (*)[4]
      |           char *
dvkasjyanov@dvkasjyanov:~/work/os/lab_prog$ gcc calculate.o main.o -o calcul -lm
```



(Рисунок 10)

В результате компиляции программа выдала ошибку.

- Исправляю синтаксические ошибки в файле **main.c**: в строке `scanf("%s", &Operation);` нужно убрать знак `&`, потому что имя массива символов уже является указателем на первый элемент этого массива (Рисунок 11).

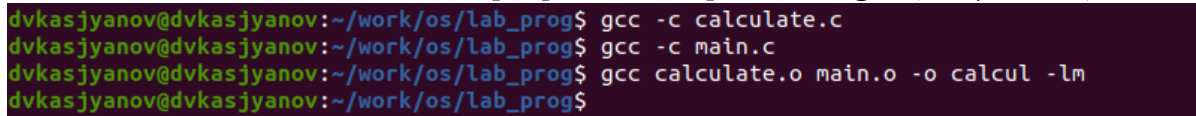


```
emacsvdvkasjyanov
File Edit Options Buffers Tools C Help
Save Undo
////////////////////////////////////
// main.c
#include <stdio.h>
#include "calculate.h"

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("%.2f\n",Result);
    return 0;
}
```

(Рисунок 11)

Снова выполняю компиляцию программы посредством **gcc** (Рисунок 12).

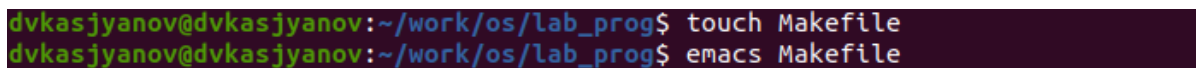


```
dvkasjyanov@dvkasjyanov:~/work/os/lab_prog$ gcc -c calculate.c
dvkasjyanov@dvkasjyanov:~/work/os/lab_prog$ gcc -c main.c
dvkasjyanov@dvkasjyanov:~/work/os/lab_prog$ gcc calculate.o main.o -o calcul -lm
dvkasjyanov@dvkasjyanov:~/work/os/lab_prog$
```

(Рисунок 12)

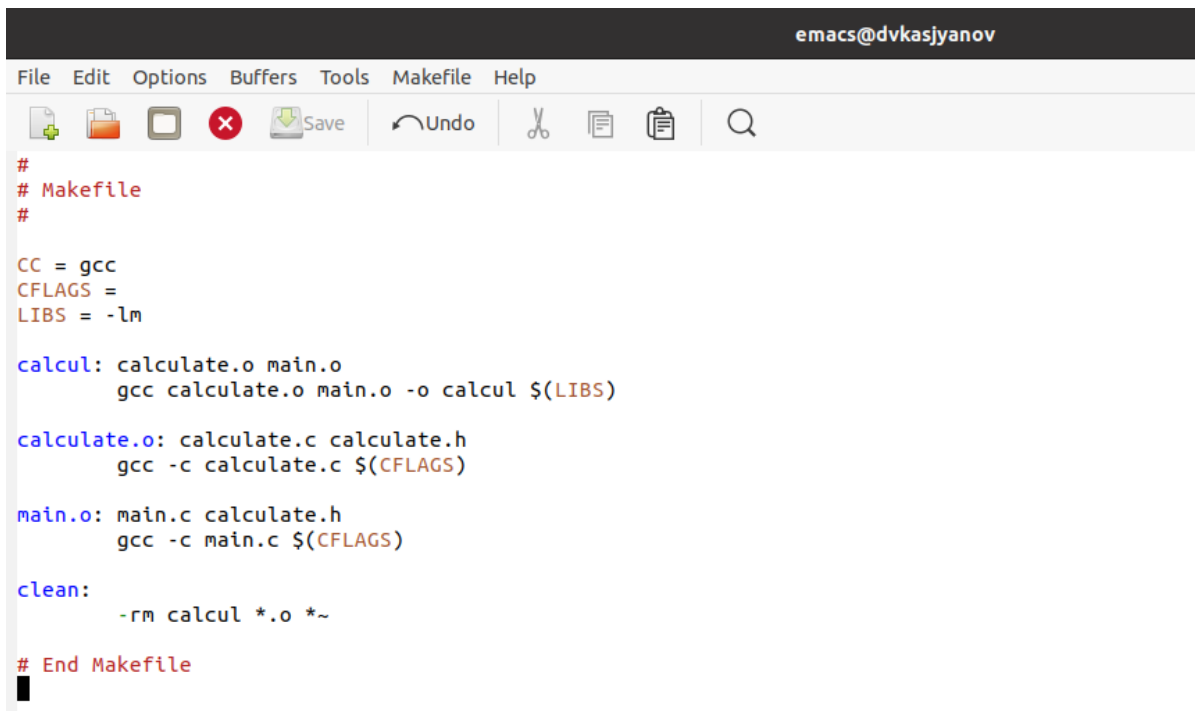
Программа работает корректно.

- Создаю **Makefile** (Рисунок 13). Переписываю в него текст программы из лабораторной работы (Рисунок 14).



```
dvkasjyanov@dvkasjyanov:~/work/os/lab_prog$ touch Makefile
dvkasjyanov@dvkasjyanov:~/work/os/lab_prog$ emacs Makefile
```

(Рисунок 13)



(Рисунок 14)

Данный **Makefile** необходим для автоматической компиляции файлов **calculate.c**, **main.c**, а также их объединения в один исполняемый файл **calcul**; **clean** автоматически удаляет объектные и исполняемые файлы. Переменная **CC** отвечает за утилиту для компиляции. Переменная **CFLAGS** отвечает за опции. Переменная **LIBS** отвечает за опции для объединения объектных файлов в один исполняемый файл.

#### 6. Исправлю **Makefile** (Рисунок 15):

- **CFLAGS** = **-g** - добавляю опцию **g**, необходимую для компиляции объектных файлов и их использования в программе отладчика **GDB**.
- Для того, чтобы утилита компиляции выбиралась с помощью переменной **CC**, заменяю **gcc** на **\$(CC)**.



(Рисунок 15)

Используя **Makefile**, удаляю исполняемые и объектные файлы (Рисунок 16), выполняю компиляцию файлов (Рисунок 17).

```
dvkasjyanov@dvkasjyanov:~/work/os/lab_prog$ make clean
rm calcul *.o *~
```

(Рисунок 16)

```
dvkasjyanov@dvkasjyanov:~/work/os/lab_prog$ make calculate.o
gcc -c calculate.c -g
dvkasjyanov@dvkasjyanov:~/work/os/lab_prog$ make main.o
gcc -c main.c -g
dvkasjyanov@dvkasjyanov:~/work/os/lab_prog$ make calcul
gcc calculate.o main.o -o calcul -lm
dvkasjyanov@dvkasjyanov:~/work/os/lab_prog$
```

(Рисунок 17)

**Makefile** работает корректно.

- Запускаю отладчик **GDB**, загрузив в него программу **calcul** для отладки (Рисунок 18):

```

dvkasjyanov@dvkasjyanov:~/work/os/lab_prog$ gdb ./calcul
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 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-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://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...

```

(Рисунок 18)

- Для запуска программы внутри отладчика ввожу команду `run`. Складываю числа 3 и 5 (Рисунок 19):

```

(gdb) run
Starting program: /home/dvkasjyanov/work/os/lab_prog/calcul
Число: 3
Операция (+, -, *, /, pow, sqrt, sin, cos, tan): +
Второе слагаемое: 5
      8.00
[Inferior 1 (process 2944) exited normally]
(gdb) 

```

(Рисунок 19)

- Для постраничного (по 10 строк) просмотра исходного код использую команду `list` (Рисунок 20):

```

(gdb) list
1      //////////////////////////////////////
2      // main.c
3
4      #include <stdio.h>
5      #include "calculate.h"
6
7      int
8      main (void)
9      {
10         float Numeral;
(gdb) 

```

(Рисунок 20)

- Для просмотра строк с 12 по 15 основного файла использую `list 12,15` (Рисунок 21):

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

(Рисунок 21)

- Для просмотра определённых строк не основного файла использую `list calculate.c:20,29` (Рисунок 22):

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

(Рисунок 22)

- Устанавливаю точку останова в файле **calculate.c** на строке номер 21 (Рисунок 23):

```
list calculate.c:20,27
break 21
```

```
(gdb) list calculate.c:20,27
20      {
21          printf("Вычитаемое: ");
22          scanf("%f",&SecondNumeral);
23          return(Numeral - SecondNumeral);
24      }
25      else if(strncmp(Operation, "*", 1) == 0)
26      {
27          printf("Множитель: ");
(gdb) break 21
Breakpoint 1 at 0x555555552dd: file calculate.c, line 21.
(gdb) █
```

(Рисунок 23)

- Вывожу информацию об имеющихся в проекте точках останова, используя `info breakpoints` (Рисунок 24):

```
(gdb) info breakpoints
Num   Type             Disp Enb Address          What
1     breakpoint       keep y  0x0000555555552dd in Calculate at calculate.c:21
(gdb)
```

(Рисунок 24)

- Запускаю программу внутри отладчика. Программа останавливается в момент прохождения точки останова (Рисунок 25):

```
(gdb) run
Starting program: /home/dvkasjyanov/work/os/lab_prog/calcul
Число: 5
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): -
Breakpoint 1, Calculate (Numeral=5, Operation=0x7fffffffde54 "-") at calculate.c:21
21      printf("Вычитаемое: ");
(gdb) backtrace
#0  Calculate (Numeral=5, Operation=0x7fffffffde54 "-") at calculate.c:21
#1  0x00005555555555bd in main () at main.c:17
(gdb) █
```

(Рисунок 25)

- Смотрю, чему равно на этом этапе значение переменной **Numeral**, используя команду `print Numeral` (Рисунок 26):

```
(gdb) print Numeral
$1 = 5
(gdb) █
```

(Рисунок 26)

На экран выводится число 5.

- Сравниваю с результатом вывода на экран после использования команды `display Numeral` (Рисунок 27):

```
(gdb) display Numeral
1: Numeral = 5
(gdb) █
```

(Рисунок 27)

Значения совпадают.

- Убираю точки останова (Рисунок 28):

```
info breakpoints
```

```
delete 1
```

```
(gdb) info breakpoints
Num      Type           Disp Enb Address            What
1        breakpoint     keep y   0x000000000000012dd in Calculate at calculate.c:21
(gdb) delete 1
(gdb) info breakpoints
No breakpoints or watchpoints.
(gdb) █
```

(Рисунок 28)

7. С помощью утилиты **splint** проанализирую коды файлов **calculate.c** и **main.c**.

```

dvkasjyanov@dvkasjyanov:~/work/os/lab_prog$ splint calculate.c
Splint 3.1.2 --- 20 Feb 2018

calculate.h:7:37: 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:10:31: Function parameter Operation declared as manifest array
                    (size constant is meaningless)
calculate.c: (in function Calculate)
calculate.c:16:7: 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:22:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:28:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:34:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:35:10: 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:38:10: Return value type double does not match declared type float:
                    (HUGE_VAL)
    To allow all numeric types to match, use +relaxtypes.
calculate.c:46:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:47:13: Return value type double does not match declared type float:
                    (pow(Numeral, SecondNumeral))
calculate.c:50:11: Return value type double does not match declared type float:
                    (sqrt(Numeral))
calculate.c:52:11: Return value type double does not match declared type float:
                    (sin(Numeral))
calculate.c:54:11: Return value type double does not match declared type float:
                    (cos(Numeral))
calculate.c:56:11: Return value type double does not match declared type float:
                    (tan(Numeral))

```

(Рисунок 29)

```

calculate.c:60:13: Return value type double does not match declared type float:
                    (HUGE_VAL)

Finished checking --- 15 code warnings

```

(Рисунок 30)

```

dvkasjyanov@dvkasjyanov:~/work/os/lab_prog$ splint main.c
Splint 3.1.2 --- 20 Feb 2018

calculate.h:7:37: 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:14:3: 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:16:3: Return value (type int) ignored: scanf("%s", Oper...

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

```

(Рисунок 31)

Выяснилось, что **calculate.c** и **main.c** возвращают некоторое целое значение.



Некоторый параметр в **main.c** является массивом из нескольких элементов, но размер массива в данном контексте игнорируется, т.к. имя массива является указателем на его первый элемент. Именно с этим связана ошибка компиляции **gcc**.

Выводится предупреждение о том, что в файле **calculate.c** происходит сравнение вещественного числа с нулем, что может привести к неожиданным результатам. Это связано со внутренним представлением чисел с плавающей запятой. Также возвращаемые значения (тип **double**) в функциях **pow**, **sqrt**, **sin**, **cos** и **tan** записываются в переменную типа **float**, что свидетельствует о потере точности и значимости.

### 3 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 расширением (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:

```
<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) Схема отладки программы показана в пункте 6.

10) В коде программы **main.c** допущена ошибка: 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

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.



## 5 Библиография

Лабораторная работа №14 - “Средства для создания приложений в ОС UNIX”

Обзор процесса разработки программного обеспечения

makefile:4: \*\*\* missing separator. Stop