

Отчёт по лабораторной работе №14

**Дисциплина: Операционные системы
Джеффри Родригес Сантос**

Содержание

1	Цель работы	5
2	Задание	6
3	Выполнение лабораторной работы	8
4	Контрольные вопросы	20
5	Выводы	25

Список таблиц

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

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`:
`gcc -c calculate.c`
`gcc -c main.c`
`gcc calculate.o main.o -o calcul -lm`
4. При необходимости исправьте синтаксические ошибки.
5. Создайте `Makefile`. Поясните в отчёте его содержание.
6. С помощью `gdb` выполните отладку программы `calcul` (перед использованием `gdb` исправьте `Makefile`):
 - Запустите отладчик `GDB`, загрузив в него программу для отладки
 - Для запуска программы внутри отладчика введите команду `run`
 - Для постраничного (по 10 строк) просмотра исходного кода используйте команду `list`
 - Для просмотра строк с 12 по 15 основного файла используйте `list` с параметром

рами

- Для просмотра определённых строк не основного файла используйте list с параметрами

- Установите точку останова в файле `calculate.c` на строке номер 21
- Выведите информацию об имеющихся в проекте точке останова
- Запустите программу внутри отладчика и убедитесь, что программа останавливается в момент прохождения точки останова
- Отладчик выдаст информацию, а команда `backtrace` покажет весь стек вызываемых функций от начала программы до текущего места
- Посмотрите, чему равно на этом этапе значение переменной `Numeral`. На экран должно быть выведено число 5
- Сравните с результатом вывода на экран после использования команды
- Уберите точки останова

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

3 Выполнение лабораторной работы

1. В домашнем каталоге создаю подкаталог calculate с помощью команды «mkdir calculate».
2. Создал в каталоге файлы: calculate.h, calculate.c, main.c, используя команды «cd calculate» и «touch calculate.h calculate.c main.c» (рис. -fig. 3.1).

```
jeffrey@jeffrey-VirtualBox:~$ mkdir calculate
jeffrey@jeffrey-VirtualBox:~$ ls
abc1      common.h~  lab009.cpp~  logfile      script1.sh  sht.places
a.txt     conf.txt   lab10.sh     Makefile     script1.sh~ snap
australia c.txt      lab10.sh~    Makefile~    script2.sh  Templates
backup    Desktop   lab14.md     may          script2.sh~ Videos
backup.sh documents  lab3.md      monthly     script3.sh  work
b.txt     downloads lab4.md      Music       script3.sh~
calculate feathers   lab91.sh     my_os       script4.sh
client.c  file.txt   lab9.txt     Pictures    script4.sh~
client.c~ helloworld laboratory   play       server.c
common.h  initial-commit laboratory2  Public     server.c~
jeffrey@jeffrey-VirtualBox:~$ cd calculate
jeffrey@jeffrey-VirtualBox:~/calculate$ touch calculate.h calculate.c main.c
jeffrey@jeffrey-VirtualBox:~/calculate$ ls
calculate.c calculate.h main.c
jeffrey@jeffrey-VirtualBox:~/calculate$
```

Рис. 3.1: Создал каталог и файлы в нём

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.

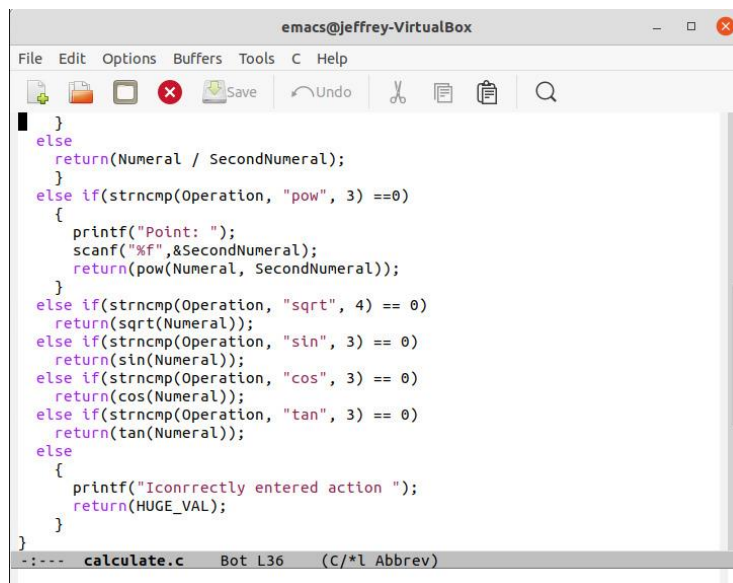
Открыв редактор Emacs, приступил к редактированию созданных файлов.

Реализация функций калькулятора в файле calculate.c (рис. -fig. 3.2)(рис. -fig. 3.3).


```
#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("The second term: ");
        scanf("%f", &SecondNumeral);
        return(Numeral + SecondNumeral);
    }
    else if(strncmp(Operation, "-", 1) == 0)
    {
        printf("Deductible: ");
        scanf("%f", &SecondNumeral);
        return(Numeral - SecondNumeral);
    }
    else if(strncmp(Operation, "*", 1) == 0)
    {
        printf("Multiplier: ");
    }
}

-:--- calculate.c Top L1 (C/*l Abbrev)
tool-bar new-file
```

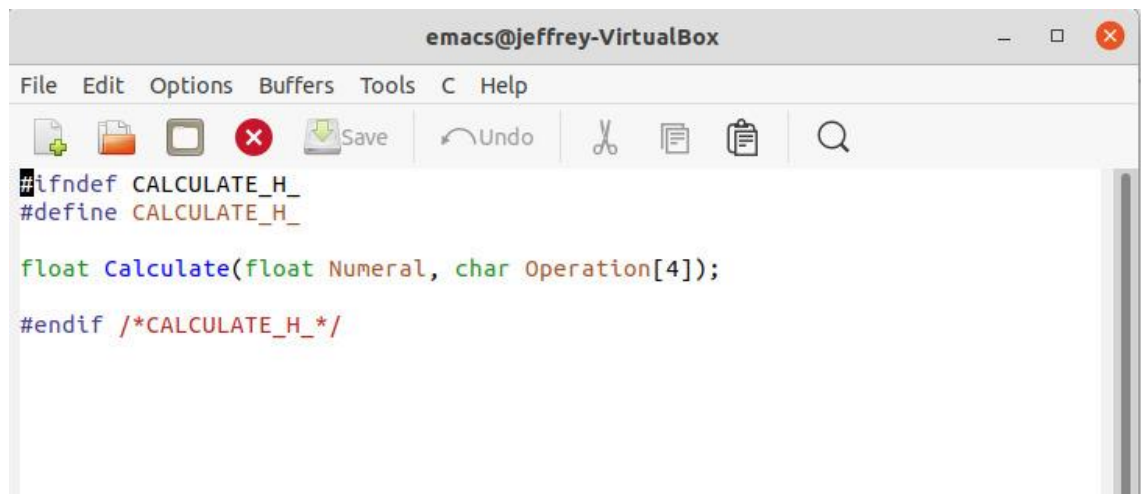


```
}
else
    return(Numeral / SecondNumeral);
}
else if(strncmp(Operation, "pow", 3) == 0)
{
    printf("Point: ");
    scanf("%f", &SecondNumeral);
    return(pow(Numeral, SecondNumeral));
}
else if(strncmp(Operation, "sqrt", 4) == 0)
    return(sqrt(Numeral));
else if(strncmp(Operation, "sin", 3) == 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("Iconrrectly entered action ");
    return(HUGE_VAL);
}
}

-:--- calculate.c Bot L36 (C/*l Abbrev)
```

Рис. 3.2: Реализация функций калькулятора в файле calculate.c

Интерфейсный файл `calculate.h`, описывающий формат вызова функции калькулятора (рис. -fig. 3.4).



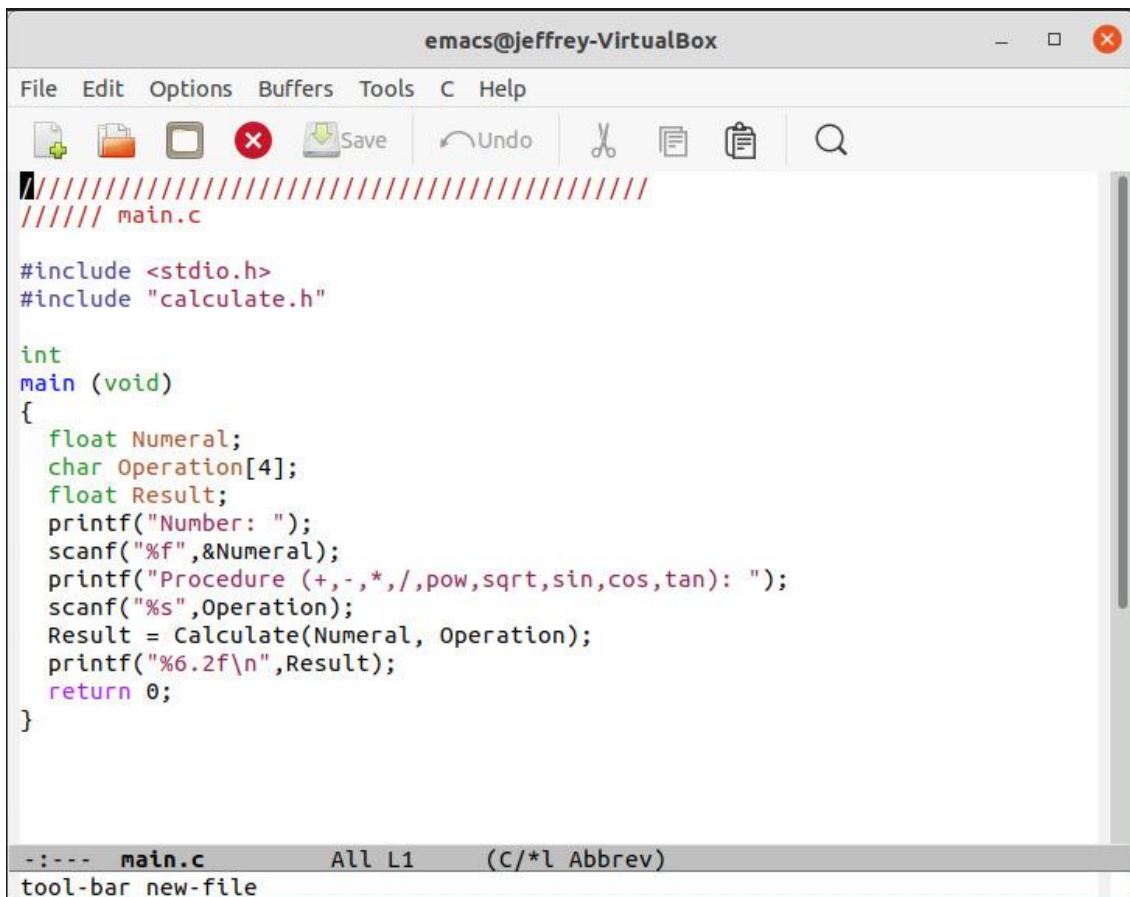
```
#ifndef CALCULATE_H_
#define CALCULATE_H_

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

#endif /*CALCULATE_H_*/
```

Рис. 3.4: Интерфейсный файл `calculate.h`

Основной файл `main.c`, реализующий интерфейс пользователя к калькулятору (рис. -fig. 3.5).



```
#####  
///// main.c  
  
#include <stdio.h>  
#include "calculate.h"  
  
int  
main (void)  
{  
    float Numeral;  
    char Operation[4];  
    float Result;  
    printf("Number: ");  
    scanf("%f",&Numeral);  
    printf("Procedure (+,-,*,/,pow,sqrt,sin,cos,tan): ");  
    scanf("%s",Operation);  
    Result = Calculate(Numeral, Operation);  
    printf("%6.2f\n",Result);  
    return 0;  
}
```

-:--- main.c All L1 (C/*l Abbrev)
tool-bar new-file

Рис. 3.5: Основной файл main.c

3. Выполнил компиляцию программы посредством gcc, используя команды «gcc -c calculate.c», «gcc -c main.c» и «gcc calculate.o main.o -o calcul -lm» (рис.-fig. 3.6).

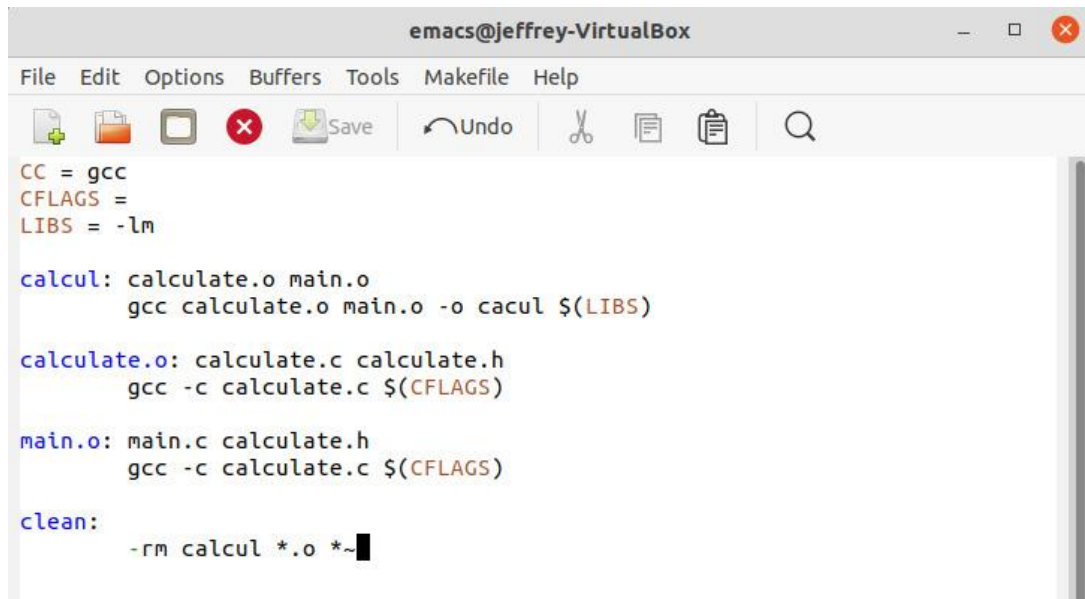
```
jeffrey@jeffrey-VirtualBox:~/work/2020-2021/09-intro/laboratory/lab_prog$ gcc -  
c main.c  
jeffrey@jeffrey-VirtualBox:~/work/2020-2021/09-intro/laboratory/lab_prog$ gcc -  
c calculate.c  
jeffrey@jeffrey-VirtualBox:~/work/2020-2021/09-intro/laboratory/lab_prog$ gcc -  
c main.c  
jeffrey@jeffrey-VirtualBox:~/work/2020-2021/09-intro/laboratory/lab_prog$ gcc c  
calculate.o main.o -o calcul -lm  
jeffrey@jeffrey-VirtualBox:~/work/2020-2021/09-intro/laboratory/lab_prog$
```

Рис. 3.6: Выполнил компиляцию программы посредством gcc

4. В ходе компиляции программы никаких ошибок выявлено не было.

Создал Makefile с необходимым содержанием (рис. -fig).

5. Данный файл необходим для автоматической компиляции файлов calculate.c (цель calculate.o), main.c (цель main.o), а также их объединения в один испол- няемый 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.



```
CC = gcc
CFLAGS =
LIBS = -lm

calcul: calculate.o main.o
    gcc calculate.o main.o -o cacul $(LIBS)

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

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

clean:
    -rm calcul *.o *~
```

Рис. 3.7: Создал Makefile с необходимым содержанием

6. Далее исправил Makefile (рис. -fig. 3.8). В переменную CFLAGS добавил опцию -g, необходимую для компиляции объектных файлов и их использования в программе отладчика GDB. Сделал так, что утилита компиляции выбирается с помощью переменной CC.

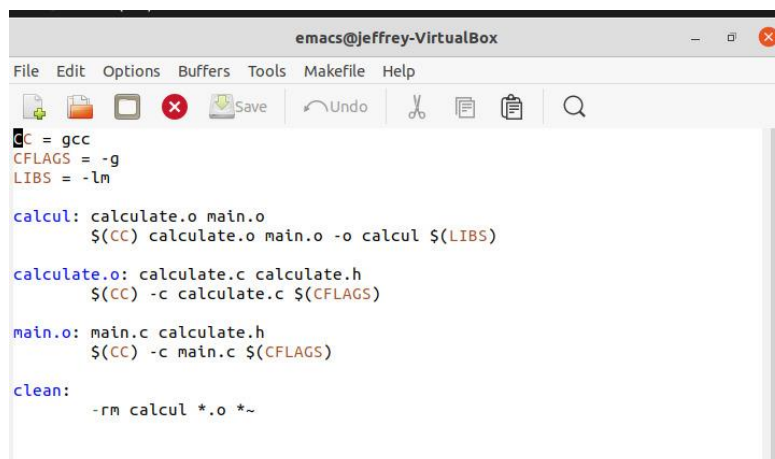


Рис. 3.8: далее исправил Makefile

После этого я удалил исполняемые и объектные файлы из каталога с помощью команды «make clean». Выполнил компиляцию файлов, используя команды «make calculate.o», «make main.o», «male calcul» (рис.fig. 3.9).

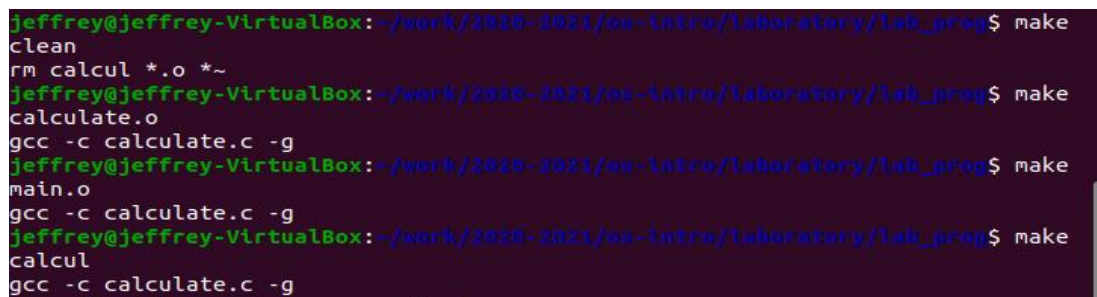


Рис. 3.9: Используем команды make

Далее с помощью gdb выполнил отладку программы calcul. Запустил отладчик GDB, загрузив в него программу для отладки, используя команду: «gdb ./calcul» (рис. -fig. 3.10).

```

jeffrey@jeffrey-VirtualBox: ~/work/2020-2021/os-intro/laboratory/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...
(gdb) run
Starting program: /home/jeffrey/work/2020-2021/os-intro/laboratory/lab_prog/calcul
Number: 6
Procedure (+,-,*,/,pow,sqrt,sin,cos,tan): +
The second term: list
        6.00
[Inferior 1 (process 8366) exited normally]
(gdb) list
1          //////////////////////////////////////////

```

Рис. 3.10: Запустил отладчик GDB

Для запуска программы внутри отладчика ввёл команду «run» (рис. -fig. 3.11).

```

[Inferior 1 (process 8366) exited normally]
(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)

```

Рис.

3.12: Использовал команду «list»

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


```

20     float Numeral;
(gdb) list
11     char Operation[4];
12     float Result;
13     printf("Number: ");
14     scanf("%f",&Numeral);
15     printf("Procedure (+,-,*,/,pow,sqrt,sin,cos,tan): ");
16     scanf("%s",Operation);
17     Result = Calculate(Numeral, Operation);
18     printf("%6.2f\n",Result);
19     return 0;
20 }
(gdb) list calculate.c:20,29
20     return(Numeral - SecondNumeral);
21 }
22     else if(strncmp(Operation, "*", 1) == 0)
23     {
24         printf("Multiplier: ");
25         scanf("%f",&SecondNumeral);
26         return(Numeral * SecondNumeral);
27     }
28     else if(strncmp(Operation, "/", 1) == 0)
29     {
(gdb) █

```

Рис. 3.13: Просмотр строк с 12 по 15

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

Рис. 3.14: Просмотр определённых строк не основного файла

Установил точку останова в файле calculate.c на строке номер 18, используя команды «list calculate.c:15,22» и «break 18» (рис. -fig. 3.15).


```

(gdb) list calculate.c:20,29
20      return(Numeral - SecondNumeral);
21      }
22      else if(strncmp(Operation, "*", 1) == 0)
23      {
24          printf("Multiplier: ");
25          scanf("%f",&SecondNumeral);
26          return(Numeral * SecondNumeral);
27      }
28      else if(strncmp(Operation, "/", 1) ==0)
29      {
(gdb) list calculate.c:20,27
20      return(Numeral - SecondNumeral);
21      }
22      else if(strncmp(Operation, "*", 1) == 0)
23      {
24          printf("Multiplier: ");
25          scanf("%f",&SecondNumeral);
26          return(Numeral * SecondNumeral);
27      }
(gdb) break 21
Breakpoint 1 at 0x455835553119: file calculate.c, line 22.
(gdb) info breakpoints
Num      Type             Disp Enb Address                  What
1        breakpoint       keep y   0x000035553119          in calculate
                                           at calculate.c:22
(gdb)

```

Рис. 3.14: Просмотр определённых строк не основного файла

Установил точку останова в файле calculate.c на строке номер 18, используя команды «list calculate.c:15,22» и «break 18» (рис. -fig. 3.15).

Вывел 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  0x0000000000000000 in Calculate
                                at calculate.c:22
(gdb) run
```

Рис. 3.16: Вывел информацию об имеющихся в проекте точках останова

Запустил программу внутри отладчика и убедился, что программа остановилась в момент прохождения точки останова. Использовал команды «run», «5», «—» и «backtrace» (рис. -fig. 3.17).

```
(gdb) run
Starting program: /home/jeffrey/work/2020-2021/os-intro/laboratory/lab_prog/calcul
Number: 5
Procedure (+,-,*,/,pow,sqrt,sin,cos,tan): -
Breakpoint 3, Calculate (Numeral=5, Operation=0x7fffffffde24 "-")
at calculate.c:18
18     printf("Deductible: ");
(gdb) backtrace
#0 Calculate (Numeral=5, Operation=0x7fffffffde24 "-") at calculate.c:18
#1 0x0000000000000000 in main () at main.c:17
```

Рис. 3.17: Запустил программу внутри отладчика до точки

останова Посмотрел, чему равно на этом этапе значение переменной

Numeral, введя

команду «print Numeral» (рис. -fig. 3.18).

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

Рис. 3.18: Посмотрел, чему равно Numeral
Сравнил 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: Сравнил с результатом вывода на экран

Убрал точку останова с помощью команд «info breakpoints» и «delete 3» (рис.-fig. 3.20).

```
(gdb) info breakpoints  
Num   Type           Disp Enb Address                What  
3      breakpoint      keep y  0x0000000000000000 in Calculate  
                                at calculate.c:18
```

Рис. 3.20: Убрал точку останова

4 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.

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 `C` 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.

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.

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.

5 Выводы

В ходе выполнения данной 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.