

Операционные системы

Лабораторная работа №14

Крупенникова Виктория Александровна

Содержание

1	Цель работы	3
2	Выполнение лабораторной работы	4
3	Контрольные вопросы:	13
4	Выводы	18

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 Выполнение лабораторной работы

- 1) В домашнем каталоге создаю подкаталог `~/work/os/lab_prog` помощью команды «`mkdir -p ~/work/os/lab_prog`»

```
vakrupennikova@dk6n66 ~ $ mkdir -p ~/work/os/lab_prog
```

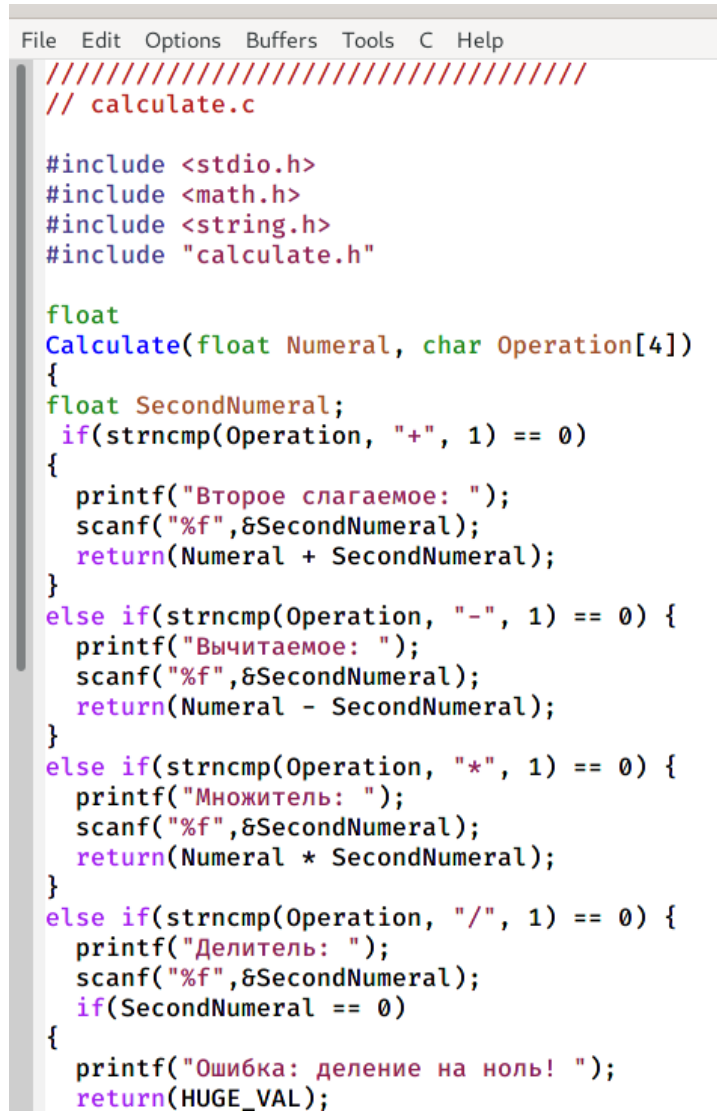
Figure 2.1: Создание подкаталога

- 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`»

```
vakrupennikova@dk6n66 ~ $ cd ~/work/os/lab_prog
vakrupennikova@dk6n66 ~/work/os/lab_prog $ touch calculate.h calculate.c main.c
vakrupennikova@dk6n66 ~/work/os/lab_prog $ ls
calculate.c calculate.h main.c
```

Figure 2.2: Создание файлов для работы

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`



```
File Edit Options Buffers Tools C Help
////////////////////////////////////
// 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);
    }
    else if(strncmp(Operation, "/", 1) == 0) {
        printf("Делитель: ");
        scanf("%f",&SecondNumeral);
        if(SecondNumeral == 0)
        {
            printf("Ошибка: деление на ноль! ");
            return(HUGE_VAL);
        }
    }
}
```

Figure 2.3: Код для калькулятора

```

} 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);
} }

```

Figure 2.4: Продолжение кода

Интерфейсный файл calculate.h, описывающий формат вызова функции калькулятора.

```

File Edit Options Buffers Tools C Help
//////////////////////////////////// // calculate.h

#ifndef CALCULATE_H_ #define CALCULATE_H_

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

#endif /*CALCULATE_H_*/

```

Figure 2.5: calculate.h

Основной файл main.c, реализующий интерфейс пользователя к калькулятору

```

File Edit Options Buffers Tools C Help
//////////////////////////////////// // 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; }

```

Figure 2.6: main.c

- 3) Выполнила компиляцию программы посредством gcc (версия компилятора: 8.3.0-19), используя команды «gcc-calculate.c», «gcc-cmain.c» и «gcccalculate.o main.o -ocalcul-lm»

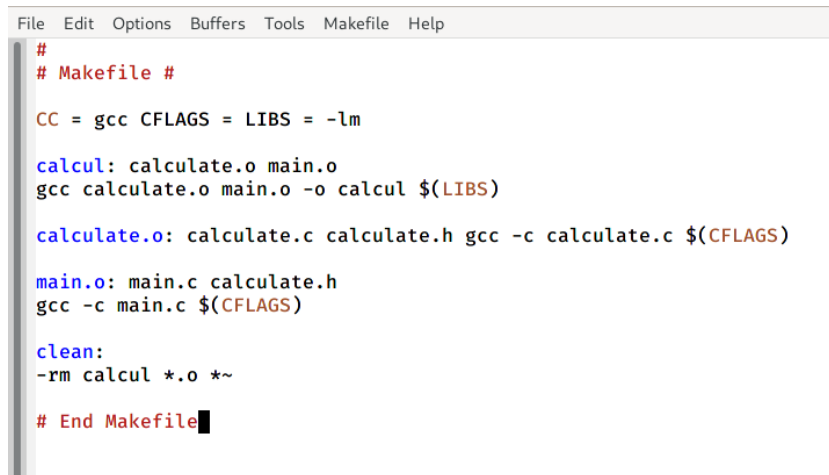
```

vakrupennikova@dk6n66 ~/work/os/lab_prog $ gcc -c calculate.c
vakrupennikova@dk6n66 ~/work/os/lab_prog $ gcc -c main.c
vakrupennikova@dk6n66 ~/work/os/lab_prog $ gcc calculate.o main.o -o calcul -lm

```

Figure 2.7: Команды gcc

- 4) В ходе компиляции программы никаких ошибок выявлено не было.
- 5) Создала Makefile с необходимым содержанием



```

File Edit Options Buffers Tools Makefile Help
#
# Makefile #

CC = gcc CFLAGS = LIBS = -lm

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

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

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

clean:
-rm calcul *.o *~

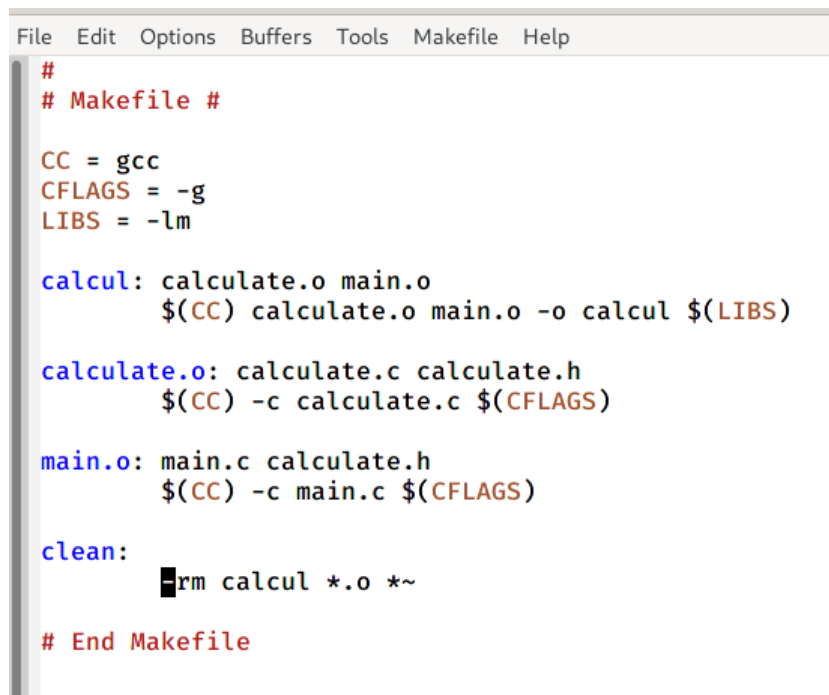
# End Makefile

```

Figure 2.8: 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



```
#
# 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 calcul *.o *~

# End Makefile
```

Figure 2.9: Измененный Makefile

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 clean».

Выполнила компиляцию файлов, используя команды «make calculate.o», «make main.o», «make calcul».

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


```

vakrupennikova@dk6n66 ~/work/os/lab_prog $ gdb ./calcul
GNU gdb (Gentoo 10.1 vanilla) 10.1
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-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://bugs.gentoo.org/>.
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...
(No debugging symbols found in ./calcul)
(gdb) run

```

Figure 2.10: Запуск GDB

Для запуска программы внутри отладчика ввела команду «run»

```

(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /afs/.dk.sci.pfu.edu.ru/home/m/m/mmbezruk/calcul
Число: 5
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): *
Множитель: 7
35.00

```

Figure 2.11: Команда «run»

Для постраничного (по 10 строк) просмотра исходного кода использовала команду «list»

```

(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) list
11     char Operation[4];
12     float Result;
13     printf("Число: ");
14     scanf("%f",&Numeral);
15     printf("Операция (+,-,*,/,pow,sqrt,sin,cos,tan): ");
16     scanf("%s",&Operation);
17     Result = Calculate(Numeral, Operation);
18     printf("%6.2f\n",Result);
19     return 0;
20 }

```

Figure 2.12: Команда «list»

Для просмотра строк с 12 по 15 основного файла использовала команду «list 12,15»

Для просмотра определённых строк не основного файла использовала команду «list calculate.c:20,29»

```

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

```

Figure 2.13: Команда «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»

```

(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 0x555555400955: file calculate.c, line 21.

```

Figure 2.14: Точка останова

Вывела информацию об имеющихся в проекте точках останова с помощью команды «info breakpoints»

```

(gdb) info breakpoints
Num   Type             Disp Enb Address            What
1     breakpoint       keep y   0x0000555555400955 in Calculate at calculate.c:21

```

Figure 2.15: Вывод информации

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

```

(gdb) run
Starting program: /afs/.dk.sci.pfu.edu.ru/home/m/m/mmbezruk/calcul
Число: 5
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): -
Breakpoint 1, Calculate (Numeral=5, Operation=0x7fffffffcea4 "-") at calculate.c:21
21      printf("Вычитаемое: ");

```

Figure 2.16: Запуск программы

Посмотрела, чему равно на этом этапе значение переменной Numeral, введя команду «print Numeral»

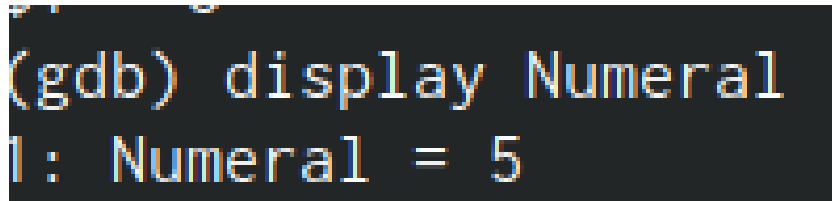
```

(gdb) print Numeral
$1 = 5

```

Figure 2.17: Переменная Numeral

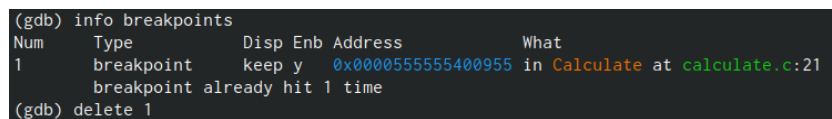
Сравнила с результатом вывода на экран после использования команды «display Numeral». Значения совпадают.



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

Figure 2.18: Сравнение

Убрала точки останова с помощью команд «info breakpoints» и «delete 1»



```
(gdb) info breakpoints
Num   Type             Disp Enb Address            What
1     breakpoint       keep y   0x000055555400955 in Calculate at calculate.c:21
      breakpoint already hit 1 time
(gdb) delete 1
```

Figure 2.19: Удаление точки останова

7) С помощью утилиты splint проанализировала коды файлов calculate.c и main.c.

Далее воспользовалась командами «splint calculate.c» и «splint main.c»

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.

3 Контрольные вопросы:

- 1) Чтобы получить информацию о возможностях программ gcc, make, gdb и др. нужно воспользоваться командой тап 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 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 -o 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 hellomain.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. 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 = gccCFLAGS =# Compile abcd.c normallyabcd: abcd.c\$(CC) -o abcd \$(CFLAGS) abcd.cclean:-rm abcd .o ~# EndMakefileforabcd.c В этом примере в начале файла заданы три переменные: CСи 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 UNIXвходит отладчик GDB (GNUDebugger).

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-cfile.c -g После этого для начала работы с gdb необходимо в командной строке ввести одноимённую команду, указав в качестве аргумента анализируемый бинарный файл: gdbfile.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 `mangdb`.

- 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 Санализатор `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.